БИБЛИОТЕКА ПРОГРАММИСТА

Эндрю Троелсен

C# и платформа .NET

- философия .NET
- основы языка C#
- интерфейсы и коллекции
- элементы управления
- ввод, вывод и сериализация объектов
- взаимодействие с унаследованным программным кодом
- web-службы

Apress™

ПИТЕР

СЕРИЯ

БИБЛИОТЕКА ПРОГРАММИСТА

ПИТЕР®

ПИТЕР®

Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Новосибирск • Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск
2004

Andrew Troelsen

ББК 32.973-018
УДК 681.3.06
Т70

Троелсен. Э.
Т70 C# и платформа .NET. Библиотека программиста. — СПб.: Питер, 2004. —796 с.: ил.

ISBN 5-318-00750-3

Основная цель этой книги — дать читателю прочные знания синтаксиса и семантики C#, а также разобрать особенности архитектуры .NET. После ее прочтения вы познакомитесь со всеми основными областями, охваченными библиотекой базовых классов C#. Для приобретения практических навыков книга содержит множество примеров, иллюстрирующих излагаемый материал.

Для работы с книгой не нужен какой-либо предварительный опыт работы с C# и платформой .NET, однако при ее написании авторы ориентировались на тех разработчиков, которые уже имеют опыт работы с одним из современных языков программирования (C++, Visual Basic, Java или каким-либо другим).

ББК 32.973-018
УДК 681.3.06

ISBN 1-893115-59-3 (англ.)

ISBN 5-318-00750-3

Краткое содержание

Содержание

Посвящается моей жене Аманде в благодарность за ее огромную поддержку, которую она оказывала мне в течение всей работы над книгой. Спасибо ей за то, что она вдохновляла меня даже тогда, когда казалось, что мне уже нечего сказать.

Предисловие

На момент написания этой книги платформа .NET и программирование на C# уже представляли собой заметное явление в мире программирования. Не хочется впадать в рекламную патетику, однако я абсолютно уверен, что платформа .NET - это Новый Мировой Порядок программирования под Windows (а в будущем, наверное, и не только под Windows).

.NET представляет собой совершенно новый способ создания распределенных, настольных и встроенных приложений. Очень важно сразу осознать, что .NET не имеет ничего общего с COM (кроме мощных средств интеграции двух платформ). Для типов .NET не нужны ни фабрики классов, ни поддержка IUnknown, ни регистрация в системном реестре. Эти основные элементы COM не скрыты — их просто больше нет.

Специально для новой платформы Microsoft разработала новый язык программирования — C#. Этот язык, как и Java, очень многое позаимствовал из C++ (особенно с точки зрения синтаксиса). Однако на C# сильно повлиял и Visual Basic 6.0. В целом можно сказать, что C# впитал в себя многое из того лучшего, что есть в самых разных языках программирования, и если у вас есть опыт работы с C++, Java или Visual Basic, то вы найдете в C# много знакомого.

Очень важно отметить, что платформа .NET является полностью независимой от используемых языков программирования. Вы можете использовать несколько .NET-совместимых языков программирования (скорее всего, вскоре их будет множество) даже в рамках одного проекта. Сразу скажем, что разобраться с самим языком C# достаточно просто. Наибольшие усилия вам потребуются, чтобы познакомиться с многочисленными пространствами имен и типами библиотеки базовых классов .NET. С этими типами (как и со своими собственными, созданными, например, на C#) вы сможете работать из любого .NET-совместимого языка.

Основная цель, которая преследовалась при создании этой книги, — дать вам прочные знания синтаксиса и семантики C#, а также особенностей архитектуры .NET. После прочтения страниц этой книги (весьма многочисленных) вы познакомитесь со всеми основными областями, охваченными библиотекой базовых классов C#. Ниже приводится краткое описание содержания каждой из глав этой книги.

Глава 1. Философия .NET

Эта глава посвящена, конечно же, основам .NET. Мы узнаем, чем .NET отличается от других распространенных технологий программирования и познакомимся с краеугольными камнями ее архитектуры: Common Language Runtime (CLR) — средой выполнения .NET, Common Type System (CTS) — единой системой типов и Common Language Specification (CLS) — спецификацией для всех .NET-совместимых языков программирования. После этого состоится наше первое знакомство с C# и мы научимся работать с компилятором командной строки (csc.exe). Конечно же, работа по созданию приложений на C# производится обычно в Visual Studio.NE Г. Ее мы также не обойдем своим вниманием.

Глава 2. Основы языка C#

Глава посвящена синтаксису языка C#. Вначале мы познакомимся со встроенными типами данных C# и конструкциями циклов и условных переходов. Кроме того, мы узнаем, как определять классы C#, что такое ссылочные и структурные типы, как проводить упаковку и распаковку, для чего нужны пространства имен, В C# существует один класс, который является базовым для всех остальных, — класс System.Object. В этой главе мы познакомимся как ним самим, так и с членами, которые наследуют от него все остальные классы C#.

Глава 3. C# и объектно-ориентированное программирование

C# — это полнофункциональный объектно-ориентированный язык, который поддерживает все три «столпа» объектно-ориентированного программирования: инкапсуляцию, наследование (с отношениями is-a и has-a) и полиморфизм (классический и для конкретного случая, ad hoc). В начале главы будет дан обзор того, что собой представляют эти основные сущности ООП и как именно C# их поддерживает. Кроме того, мы разберемся с созданием свойств классов, с применением ключевого слова readonly и разработкой иерархий классов. В последней части этой главы будет рассмотрен официальный и правильный способ управления любыми аномалиями времени выполнения: структурированная обработка исключений (Structured Exception Handling, SEH). Мы также рассмотрим вопросы управления оперативной памятью в .NET, службу сборщика мусора и то, как ею можно управлять при помощи типов из пространства имен System.GC.

Глава 4. Интерфейсы и коллекции

Как и в Java, и COM, в C# можно использовать интерфейсы (а можно и не использовать). В этой главе мы познакомимся с ролью интерфейсов, а также с тем, как их можно определить и реализовать в C#. После этого мы займемся классами, которые поддерживают несколько интерфейсов, и узнаем, как получать ссылку на интерфейс из объекта класса. Вторая часть этой главы посвящена встроенным ин-

терфейсам библиотеки базовых классов .NET и описанию того, как их можно использовать для разных целей — например, для создания собственных объектов-контейнеров. Мы также выясним, как можно создавать клонируемые типы и типы, к которым можно применять конструкцию foreach.

Глава 5. Дополнительные возможности классов C#

Как понятно из названия, в этой главе рассмотрены дополнительные возможности классов C# — более сложные, но очень важные. Мы рассмотрим применение индексаторов, перегрузку операторов, протокол событий .NET, делегаты и сами события. В .NET предусмотрена возможность автоматического составления документации в формате XML к вашим программным модулям. Эти возможности также будут рассмотрены в этой главе.

Глава 6. Сборки, потоки и домены приложений

К этому моменту у вас уже будет определенный опыт создания приложений C#. В этой главе мы узнаем, как наделить их новыми возможностями: как создать приложение, состоящее не из одного файла EXE, а из исполняемого файла и библиотек кода, как выглядит сборка .NET «изнутри», чем различаются «частные» сборки и сборки для общего пользования и т. п. Мы познакомимся с глобальным кэшем сборок (Global Assembly Cache, GAC), который есть на каждом компьютере со средой выполнения .NET, и с файлами конфигурации в формате XML. Чтобы лучше осознать возможности CLR, мы создадим примеры с применением межъязыкового наследования и многопоточные приложения.

Глава 7. Рефлексия типов и программирование с использованием атрибутов

Рефлексия — это процесс получения информации о типах прямо во время выполнения программы. Для этого в .NET предназначены типы из пространства имен System.Reflection. Мы узнаем вначале, как получать информацию о содержимом сборки «на лету», а потом — как создавать сборки «на лету» при помощи пространства имен System.Reflection.Emit. Кроме того, в этой главе мы познакомимся с поздним связыванием в C# и ситуациями, когда применение этой технологии необходимо. Мы также рассмотрим применение атрибутов при создании типов и узнаем, как можно использовать атрибуты для создания метаданных (генерируемых компилятором) для хранения информации о приложении.

Глава 8. Окна становятся лучше: введение в Windows.Forms

Несмотря на название, в .NET реализованы замечательные средства для создания традиционных настольных приложений. В этой главе мы узнаем, как можно со-

здавать главную форму (окно) приложения при помощи типов из пространства имен System.Windows.Forms. После этого мы познакомимся с тем, как производится создание ниспадающих и контекстных меню, панелей инструментов и строк состояния. Кроме того, в этой же главе мы рассмотрим приемы взаимодействия с реестром операционной системы и журналом событий Windows 2000.

Глава 9. Графика становится лучше (GDI+)

Эта глава, как понятно из названия, посвящена работе с графикой. Мы узнаем, как можно выводить геометрические фигуры, растровые изображения и текстовую информацию (графическими средствами) на формах Windows. Мы научимся перетаскивать элементы и проверять, попал ли указатель мыши при щелчке в определенную область на форме или нет. В последней части главы мы рассмотрим формат ресурсов .NET, основанный на синтаксисе XML.

Глава 10. Элементы управления

Это — третья и последняя глава, посвященная созданию приложений Windows Forms. Она полностью посвящена вопросам, связанным с применением в графических приложениях Windows разнообразных элементов управления. Помимо стандартных элементов управления (таких как TextBox, Button, ListBox и т. п.) мы познакомимся и с более экзотическими видами, такими как Calendar и DataGrid. В последней части главы мы будем учиться создавать пользовательские диалоговые окна и применять технологию, называемую наследованием форм.

Глава 11. Ввод, вывод и сериализация объектов

В .NET, конечно же, предусмотрен полный набор типов, чтобы максимально облегчить реализацию любых действий, связанных с вводом-выводом. Эти типы (рассматриваемые в этой главе) позволяют производит чтение и запись в файл, область в оперативной памяти, буфер обмена и т. п. Очень важную роль играют также службы сериализации .NET. Сериализация — это процесс записи объекта в постоянное хранилище с сохранением всей его внутренней информации о состоянии, Сериализация обычно производится для целей последующего восстановления объекта (десериализации). В .NET предусмотрены средства, которые позволяют сериализовать объекты в двух форматах: в двоичном и в формате XML. Все эти возможности рассмотрены на примерах в этой главе.

Глава 12. Взаимодействие с унаследованным программным кодом

Как ни удивительно, но после появления .NET такая технология, как Component Object Model (COM) может с полным правом называться унаследованной. Как мы уже говорили, архитектура .NET не имеет практически ничего общего с COM.

Однако в .NET предусмотрены мощные средства обеспечения взаимодействия между этими программными технологиями: возможность обращения из клиента .NET к COM-серверу (посредством прокси-сборки и служб RCW) и возможность обращения клиента COM к сборке .NET, притворяющейся COM-сервером (при помощи промежуточного модуля и служб CCW). Кроме того, в этой главе рассмотрены средства для взаимодействия .NET напрямую с Win32 API и пользовательскими модулями DLL, написанными на C, а также вопросы создания типов .NET, предназначенных для работы под управлением среды выполнения COM+.

Глава 13. Доступ к данным при помощи ADO.NET

ADO.NET имеет не так много общего с классическим ADO. ADO.NET прежде всего предназначен для работы с базами данных без необходимости использования постоянных надежных подключений. Объекта Recordset больше нет — вместо него в ADO.NET используется объект DataSet, представляющий набор таблиц (объектами DataTables) со всеми положенными отношениями между ними. Вначале мы научимся создавать и заполнять данными объект DataSet и устанавливать отношения между его внутренним таблицами, а затем познакомимся с управляемыми провайдерами — наборами готовых типов для установления соединений с базами данных SQL Server и Access. Мы научимся не только производить операции по вставке, удалению и изменению данных в источнике, но и использовать бизнес-логику, реализованную в источнике данных в виде хранимых процедур.

Глава 14. Разработка web-приложений и ASP.NET

Эта глава начинается с описания основных элементов, используемых при создании web-приложений: как можно создать интерфейс web-приложения для клиента при помощи HTML, как можно реализовать проверку вводимых пользователем данных при помощи клиентских браузерных скриптов, как реализовать ответ на запрос пользователя из web-приложения с использованием классических ASP. Однако большая часть этой главы посвящена созданию приложений ASP.NET. Мы рассмотрим основные элементы архитектуры ASP.NET и познакомся с основными приемами создания web-приложений, такими как элементы управления Web Controls, обработка событий клиентских элементов управления на сервере, использование свойств объекта Page (в том числе Request и Response).

Глава 15. Web-службы

Последняя глава этой книги посвящена созданию web-служб ASP.NET. Web-службу можно рассматривать как сборку .NET, обращение к которой производится по стандартному Интернет-протоколу HTTP. В .NET реализован целый набор специальных технологий (WSDL, SOAP, службы обнаружения), которые предназначены для обеспечения возможности реагирования web-службы на запросы со стороны клиента. Создатели .NET позаботились не только о том, чтобы мы могли создавать web-службы средствами привычных нам языков программирования

(в отличие от классических ASP), но и об удобстве создания клиентов. Клиентам web-служб совершенно не обязательно работать непосредственно с кодом XML (форматом, в котором происходит обмен данными между web-службой и клиентом) — можно при помощи специальных средств сгенерировать специальную прокси-сборку, которая возьмет на себя эту обязанность.

Что потребуется при работе с этой книгой

Во-первых, нам потребуется Visual Studio.NET. При написании этой книги ми использовали Visual Studio.NET Beta 2, и все примеры этой книги проверены именно под этой версией .NET. Конечно, вы можете использовать и свободно распространяемый компилятор C# (csc.exe) вместе с Notepad, но усилий при этом вам придется затратить гораздо больше.

Во-вторых, мы настоятельно рекомендуем вам загрузить исходный код примеров, приведенных в книге с сайта издательства «Питер» (www.piter.com). В тексте постоянно будут встречаться ссылки на возможность воспользоваться кодом этих примеров.

В-третьих, вы должны иметь определенный опыт программирования. Для работы с этой книгой не нужен какой-либо предварительный опыт работы с C# и платформой .NET, однако при ее написании мы ориентировались на тех разработчиков, которые уже имеют опыт работы с одним из современных языков программирования (C++, Visual Basic, Java или каким-либо другим).

Несмотря на весьма значительный объем, эта книга не претендует на то, что в ней будет рассказано абсолютно все о C# и платформе .NET в целом. После изучения всех пятнадцати глав вы будете готовы создавать серьезные приложения ни. C#, однако электронная документация к Visual Studio и .NET SDK останется вашим очень важным помощником, при помощи которого мы сможете получить дополнительную информацию по темам, рассмотренным в этой книге.

А теперь приступим! Я очень надеюсь, что эта книга станет надежным проводником в мире .NET.

Эндрю Троелсен (Andrew Troelsen),
Миннеаполис, Миннесота.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на web-сайте издательстваhttp://www.piter.com.

Благодарности

Эта книга никогда бы не увидела свет, если бы я не встретил замечательных людей из Apress. Прежде всего, большое спасибо Гарри Корнеллу (Gary Cornell) за его постоянную поддержку в течение всей работы над книгой. Также спасибо Грэйс Вонг (Grace Wong), которая не давала нам забывать о нашей конечной цели. Я признателен Стефани Родригес (Stephanie Rodriguez), которая провела фантастическую работу по маркетингу этой книги, рекламируя ее и рассылая пробные главы. Я особенно благодарен ей за то безграничное терпение, которое она проявляла, когда я что-нибудь забывал. Конечно, я выражаю огромную благодарность Нэнси Гюнтер (Nancy Guenther), которая сумела не только прочитать эту книгу, но и очень быстро составить к ней алфавитный указатель.

Очень большая заслуга в появлении этой книги принадлежит всем редакторам, которые над ней работали: Дорис Вонг (Doris Wong), Энн Фридман (Anne Friedman), Нэнси Рапопорт (Nancy Rapoport) и Беверли Мак-Гайр (Beverly McGuire). Отдельно хочу поблагодарить Энн, которая работала со мной чуть ли не круглые сутки, чтобы мы могли уложиться в срок.

Хочу выразить сердечную признательность нашему главному техническому редактору Эрику Гуннерсону (Eric Gunnerson) (сотруднику Microsoft и признанному гуру C#), который потратил часть своего бесценного времени, чтобы сделать эту книгу максимально точной. Большое спасибо и другим сотрудникам Microsoft — Джо Нэйлевбау (Joe Nalewabau), Нику Ходаппу (Nick Hodapp) и Дэнису Ангелайну (Dennis Angeline), которые помогали мне разобраться в возникающих проблемах. Если вам встретятся в книге какие-либо ошибки — знайте: они на моей совести (и только на моей),

И огромная благодарность моим славным товарищам по Intertech, Inc: Стиву Клоузу (Steve Close), Джине Мак-Ги (Gina McGhee), Эндрю «Командору» Зондгеросу (Andrew «Gunner» Sondgeroth) и Тому Бэрнаби (Tom Barnaby), который, будучи предельно занят работой над своей собственной книгой, все-таки находил возможность помогать мне. Еще спасибо Тому Салонеку (Tora Salonek) — за первую чашку кофе, которую он купил мне пять лет назад.

1 Философия .NET

Любому современному программисту, который желает идти в ногу с последними веяниями, каждые несколько лет приходится переучиваться. Языки (C++, Visual Basic, Java), библиотеки (MFC, ATL, STL), архитектуры (COM, CORBA), которые стали вехами в развитии программирования за последние годы, постепенно уходят в тень лучших или по крайней мере более молодых программных технологий. Вне зависимости от того, нравится это программистам или нет, этот процесс неизбежен. Платформа .NET компании Microsoft — это следующая волна коренных изменений, которая идет к нам из Редмонда.

В этой главе мы рассмотрим основополагающие понятия .NET, к которым мы затем будем обращаться во всех Остальных частях книги. Глава начинается с рассмотрения элементов, на которых основана платформа .NET — сборок (assemblies), промежуточного языка (intermediate language, IL) и компиляции в процессе выполнения (just in time compilation, JIT). Мы также рассмотрим взаимосвязи компонентов платформы .NET — Common Language Runtime (CLR), Common Type System (CTS) и Common Language Specification (CLS).

В этой главе также приведен общий обзор возможностей, которые предоставляют библиотеки базовых классов .NET и утилит (таких как ILDasm.exe), которые помогут вам в работе с этими библиотеками. В самом конце главы мы познакомимся с возможностями компиляции приложений C# с использованием компилятора командной строки (csc.exe) и интегрированной среды разработки (Integrated Development Environment, IDE) Visual Studio.NET.

Современное состояние дел

Перед тем как мы вступим в пределы вселенной .NET, полезно будет разобраться с простым вопросом — а зачем, собственно говоря, она нужна? Для этого мы очень кратко рассмотрим те технологии, которые имеются в распоряжении программистов в настоящий момент, их возможности и ограничения, а затем перейдем к тем преимуществам, которые предоставляют C# и платформа .NET в целом.

Как живут программисты, использующие Win32/C

Изначально под программированием под Windows подразумевалось программирование на C с использованием Windows Application Programming Interface (интерфейсом прикладного программирования Windows, в 32-разрядных версиях Windows — Win32 API). С использованием этой технологии было создано множество вполне достойных приложений, однако вряд ли кто-нибудь будет спорить с тем, что написание приложения с использованием только Windows API — это очень трудоемкая задача.

Еще одна проблема заключается в том, что C — достаточно суровый по отношению к программисту язык. Тем, кто создает на нем свои приложения, приходится вручную заниматься управлением памятью, выполнять расчеты при использовании указателей и работать с совершенно неестественными с точки зрения человеческого языка синтаксическими конструкциями. Кроме того, в C, конечно, недостаточно возможностей для объектно-ориентированного программирования. Если вам приходилось связывать тысячи глобальных функций Win32 API в единые гигантские конструкции, вы не будете сильно удивляться тому, что в подобных приложениях ошибки встречаются очень часто.

Как живут программисты, использующие C++/MFC

C++ — это огромный шаг вперед в отношении новых возможностей по сравнению с исходным языком C. Во многих ситуациях C++ вполне допустимо представить как объектно-ориентированную надстройку над C. Такая надстройка позволяет использовать преимущества «столпов* объектно-ориентированного программирования — инкапсуляции, полиморфизма и наследования. Однако программисты, использующие C++, остаются незащищенными от многих и часто опасных особенностей C (теми же самыми низкоуровневыми возможностями работы с памятью и трудными для восприятия синтаксическими конструкциями).

Существует множество библиотек для C++, основное назначение которых — облегчить написание приложений под Windows, предоставив для этой цели уже готовые классы. Одна из наиболее распространенных библиотек — это MFC (Microsoft Foundation Classes). MFC — это дополнительный уровень над Win32 API, который значительно упрощает работу программиста за счет использования готовых классов, макросов и мастеров. Однако MFC — это лишь частичное решение проблемы. Даже при использовании MFC программисту приходится работать со сложным для чтения кодом, весьма опасным с точки зрения возможных ошибок.

Как живут программисты, использующие Visual Basic

Люди всегда стремятся сделать свою жизнь проще. Повинуясь этому стремлению многие программисты на C++ обратили свои взоры к гораздо более простому и дружелюбному языку, каким является Visual Basic (VB). Visual Basic позволяет работать с достаточно сложными элементами интерфейса пользователя, библиотеками кода (например, COM-серверами) и средствами доступа к данным при минимальных затратах времени и сил. Visual Basic в гораздо большей степени, чем MFC, прячет от пользователя вызовы Win32 API и предоставляет большой набор интегрированных средств быстрой разработки.

Однако у Visual Basic есть и недостатки. Главный из них — это гораздо меньшие возможности, которые предоставляет этот язык, по сравнению с C++ (это утверждение справедливо, по крайней мере, для версий более ранних, чем VB.NET). Visual Basic — это язык «для работы с объектами», а не объектно-ориентированный язык в обычном понимании этого слова. В Visual Basic нет классического наследования, нет поддержки создания параметризованных классов, нет собственных средств создания многопоточных приложений — и этот список можно продолжать еще долго.

Как живут программисты, использующие Java

Язык программирования Java — это полностью объектно-ориентированный язык, который в отношении синтаксиса многое унаследовал от C++. Конечно, преимущества Java далеко не исчерпываются межплатформенностью. Язык Java в синтаксическом отношении проще и логичнее, чем C++. Java как платформа предоставляет в распоряжение программистов большое количество библиотек (пакетов), в которых содержится большое количество описаний классов и интерфейсов на все случаи жизни. С их помощью можно создавать стопроцентные приложения Java с возможностью обращения к базам данных, поддержкой передачи почтовых сообщений, *с* клиентской частью, которой необходим только web-браузер, или наоборот, с клиентской частью, обладающей изощренным интерфейсом.

Java — это очень элегантный и красивый язык. Однако при его использовании проблем также избежать не удастся. Одна из серьезных проблем заключается в том, что при создании сложного приложения на Java вам придется использовать только этот язык для создания всех частей этого приложения. В Java предусмотрено не так уж много средств для межъязыкового взаимодействия (что понятно ввиду предназначения Java быть единым многоцелевым языком программирования). В реальном мире существуют миллионы строк готового кода, которые хотелось бы интегрировать с новыми приложениями на Java. Однако это сделать очень трудно.

Java — это далеко не идеальный язык во многих ситуациях. Простой пример — если вы попытаетесь создать только на Java приложение, активно работающее с 3D-графикой, скорее всего, вы обнаружите, что работать такое приложение будет не очень быстро. Немного подумав, вы можете прийти к выводу, что для работы с 3D-графикой лучше использовать код, написанный на языке с более развитыми низкоуровневыми возможностями (например, на C++). Однако интегрировать такой код с кодом на Java вам будет очень сложно. Поскольку возможности для обращения к API компонентов, созданных на других языках, в Java очень ограничены, говорить о реальном межъязыковом взаимодействии на основе Java не приходится.

Как живут COM-программисты

Современное состояние дел таково, что если вы не строите Java-приложения, то велика вероятность, что вы осваиваете технологию Microsoft Component Object Model (COM). COM-технология провозглашает: «Если вы создаете классы в точном соответствии с требованиями COM, то у вас получится блок *повторно используемого* программного кода».

Прелесть двоичного COM-сервера заключается в том, что к нему можно обращаться из любого языка. Например, программисты, использующие C++, могут со-

здавать классы, которые можно будет использовать из приложения на VBasic. Программисты, использующие Delphi, могут использовать классы, созданные на C и т. д. Однако в межъязыковом взаимодействии COM есть свои ограничения. Например, вы не можете произвести новый тип COM от существующего (то есть не можете использовать классическое наследование). Для повторного использования существующих типов COM вам придется использовать другие, гораздо менее надежные и эффективные средства.

Большое преимущество COM заключается в том, что программист может не заботиться о физическом местонахождении компонентов. Такие средства, как Application Identifiers (AppIDs, идентификаторы приложений), стабы (stubs), прокси, среда выполнения COM, позволяют избегать при обращении к компонентам по сети необходимости помещать в приложение код для работы с сокетами, вызовами RPC и прочими низкоуровневыми механизмами. Достаточно посмотреть на такой код на Visual Basic 6.0 для клиента COM:

```
'Этот код на VB 6.0 предназначен для активации класса COM.
'созданного на любом языке. Класс COM может быть расположен
'в любом месте на локальном компьютере или в сети.
Dim c as New MyCOMClass 'Местонахождение класса 'определяется через AppID c.DoSomeWork
```

Объектная модель COM используется очень широко. Однако внутреннее устройство компонентов весьма сложно. Чтобы научиться разбираться в нем, вам придется потратить по крайней мере несколько месяцев. Написание приложений с использованием COM-компонентов можно упростить, используя стандартные библиотеки, например библитеку Active Template Library (ATL) со своим набором готовых классов, шаблонов и макросов.

Некоторые языки (например, Visual Basic) также позволяют скрыть сложность инфраструктуры COM. Однако всех сложностей избежать все равно не удастся. Например, даже если вы работаете с относительно простым и поддерживающим COM Visual Basic, вам придется решать не всегда простые вопросы, связанные с регистрацией компонентов на компьютере и развертыванием приложений.

Как живут программисты, использующие Windows DNA

Картина будет неполной, если мы не примем во внимание такую мелочь, как Интернет. За несколько последних лет Microsoft добавила в свои операционные системы большое количество средств для работы с этой средой, в том числе и средства, призванные помочь в создании Интернет-приложений. Однако построение законченного web-приложения с использованием технологии Windows DNA (Distributed iNternet Architecture — распределенная межсетевая архитектура) до сих пор остается весьма сложной задачей.

Значительная часть сложностей возникает оттого, что Windows DNA требует использования разнородных технологий и языков (ASP, HTML, XML, JavaScript, VBScript, COM(+), ADO и т. д.). Одна из проблем заключается в том, что синтаксически все эти языки и технологии очень мало похожи друг на друга. Например, синтаксис JavaScript больше похож на синтаксис C, в то время как VBScript является подмножеством Visual Basic. COM-серверы, предназначенные для работы в среде выполнения COM+, созданы на основе совершенно иных подходов, нежели ASP-страницы, которые к ним обращаются. Конечным результатом является пугающее смеше-

ние технологий. Помимо всего прочего, в каждом языке, который входит в состав Windows DNA, предусмотрена своя система типов, что также не является источником большой радости для программистов. Например, тип данных int в JavaScript — это не то же самое, что int в C, который, в свою очередь, отличен от integer в Visual Basic.

Решение .NET

На этом мы будем считать обращение к новейшей истории программирования законченным. Главный вывод, с которым вряд ли кто-нибудь будет спорить, таков: тяжела жизнь Windows-программиста. На этом фоне возможности, предлагаемые платформой .NET, позволяют радикально облегчить нашу жизнь. Один из главных принципов .NET звучит так; «Изменяйте все, что хотите, откуда вам угодно». .NET — это совершенно новая модель для создания приложений под Windows (а в будущем, видимо, и под другими операционными системами). Вот краткое перечисление основных возможностей .NET:

- Полные возможности взаимодействия с существующим кодом. Вряд ли кто-нибудь будет спорить, что это — вещь очень хорошая. Как мы увидим в главе 12, существующие двоичные компоненты COM отлично работают вместе с двоичными файлами .NET.
- Полное и абсолютное межъязыковое взаимодействие. В отличие от классического COM, в .NET поддерживаются межъязыковое наследование, межъязыковая обработка исключений и межъязыковая отладка.
- Общая среда выполнения для любых приложений .NET, вне зависимости от того, на каких языках они были созданы. Один из важных моментов при этом — то, что для всех языков используется один и тот же набор встроенных типов данных.
- Библиотека базовых классов, которая обеспечивает сокрытие всех сложностей, связанных с непосредственным использованием вызовов API, и предлагает целостную объектную модель для всех языков программирования, поддерживающих .NET.
- Про пугающую сложность COM можно забыть! IClassFactory, IUnknown, код IDL и проклятые VARIANT-совместимые типы данных (BSTR, SAFEARRAY и остальные) больше не встретятся вам в коде программ .NET.
- Действительное упрощение процесса развертывания приложения. В .NET нет необходимости регистрировать двойные типы в системном реестре. Более того, .NET позволяет разным версиям одного и того же модуля DLL мирно сосуществовать на одном компьютере.

Строительные блоки .NET (CLR, CTS и CLS)

Технологии CLR, CTS и CLS очень важны для понимания смысла платформы .NET. С точки зрения программиста .NET вполне можно рассматривать просто как новую среду выполнения и новую библиотеку базовых классов. Среда выполнения .NET как раз и обеспечивается с помощью Common Language Runtime (CLR,

стандартная среда выполнения для языков). Главная роль CLR заключается в том, чтобы обнаруживать и загружать типы .NET и производить управление ими в соответствии с вашими командами. CLR берет на себя всю низкоуровневую работу — например, автоматическое управление памятью, межъязыковым взаимодействием, развертывание (с отслеживанием версий) различных двоичных библиотек.

Еще один строительный блок платформы .NET — это Common Type System (CTS, стандартная система типов). CTS полностью описывает все типы данных, поддерживаемые средой выполнения, определяет, как одни типы данных могут взаимодействовать с другими и как они будут представлены в формате метаданных .NET (подробнее о метаданных будет рассказано ниже в этой главе).

Важно понимать, что не во всех языках программирования .NET обязательно должны поддерживаться все типы данных, которые определены в CTS. Common Language Specification (CLS) — это набор правил, определяющих подмножество общих типов данных, в отношении которых гарантируется, что они безопасны при использовании во всех языках .NET. Если вы создаете типы .NET с использованием только тех возможностей, которые совместимы с CLS, тем самым вы сделаете их пригодными для любых языков .NET.

Библиотека базовых классов .NET

Помимо спецификаций CLR и CTS/CLS платформа .NET предоставляет в ваше распоряжение также библиотеку базовых классов, доступную из любого языка программирования .NET. Библиотека базовых классов не только прячет обычные низкоуровневые операции, такие как файловый ввод-вывод, обработка графики и взаимодействие с оборудованием компьютера, но и обеспечивает поддержку большого количества служб, используемых в современных приложениях.

В качестве примера можно привести встроенные типы для обращения к базам данных, работы с XML, обеспечения безопасности при работе приложения, создания приложений для работы в Web (конечно, обеспечивается и поддержка обычных консольных и оконных приложений). С концептуальной точки зрения отношения между уровнем среды выполнения и библиотекой базовых классов .NET выглядят так, как показано на рис. 1.1.

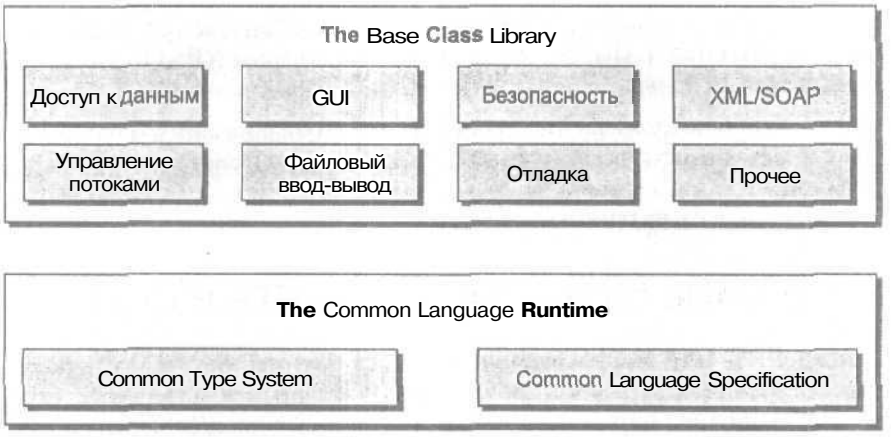


Рис. 1.1. Отношения между средой выполнения и библиотекой базовых классов .NET

Преимущества C#

Специально для платформы .NET Microsoft был разработан новый язык программирования C#. C# — это язык программирования, синтаксис которого очень похож на синтаксис Java (но не идентичен ему). Например, в C# (как в Java) определение класса состоит из одного файла (*.cs), в отличие от C++, где определение класса разбито на заголовок (*.h) и реализацию (*.cpp). Однако называть C# клоном Java было бы неверно. Как C#, так и Java основаны на синтаксических конструкциях C++. Если Java во многих отношениях можно назвать очищенной версией C++, то C# можно охарактеризовать как очищенную версию Java.

Синтаксические конструкции C# унаследованы не только от C++, но и от Visual Basic. Например, в C#, как и в Visual Basic, используются свойства классов. Как C++, C# позволяет производить перегрузку операторов для созданных вами типов (Java не поддерживает ни ту, ни другую возможность). C# — это фактически гибрид разных языков. При этом C# синтаксически не менее (если не более) чист, чем Java, так же прост, как Visual Basic, и обладает практически той же мощью и гибкостью, что и C++. Подводя итоги, еще раз выделим основные особенности C#.

- Указатели больше не нужны! В программах на C#, как правило, нет необходимости в работе с ними (однако если вам это потребуется, пожалуйста, — возможности для работы с указателями в вашем распоряжении).
- Управление памятью производится автоматически.
- В C# предусмотрены встроенные синтаксические конструкции для работы с перечислениями, структурами и свойствами классов.
- В C# осталась возможность перегружать операторы, унаследованные от C++. При этом значительная часть возникавших при этом сложностей ликвидирована.
- Предусмотрена полная поддержка использования программных интерфейсов. Однако в отличие от классического COM применение интерфейсов — это не единственный способ работы с типами, используя различные двоичные модули. .NET позволяет передавать объекты (как ссылки или как значения) через границы программных модулей.
- Также предусмотрена полная поддержка аспектно-ориентированных программных технологий (таких как атрибуты). Это позволяет присваивать типам характеристики (что во многом напоминает COM IDL) для описания в будущем поведения данной сущности.

Возможно, самое важное, что необходимо сказать про язык C#, — это то, что он генерирует код, предназначенный для выполнения только в среде выполнения .NET. Например, вы не сможете использовать C# для создания классического COM-сервера. Согласно терминологии Microsoft код, предназначенный для работы в среде выполнения .NET, — это *управляемый код* (managed code). Двоичный файл, который содержит управляемый файл, называется *сборкой* (assembly). Подробнее об этом будет сказано ниже.

Языки программирования .NET

Во время анонса платформы .NET на конференции 2000 Professional Developers Conference (PDC) докладчики называли фирмы, работающие над созданием .NET-версий своих компиляторов. На момент написания этой книги компиляторы для создания .NET-версий приложений разрабатывались более чем для 30 различных языков. Помимо четырех языков, поставляемых с Visual Studio.NET (C#, Visual Basic.NET, «Managed C++» и JScript.NET), ожидаются .NET-версии Smalltalk, COBOL, Pascal, Python, Perl и множества остальных известных языков программирования. Общая картина представлена на рис. 1.2.

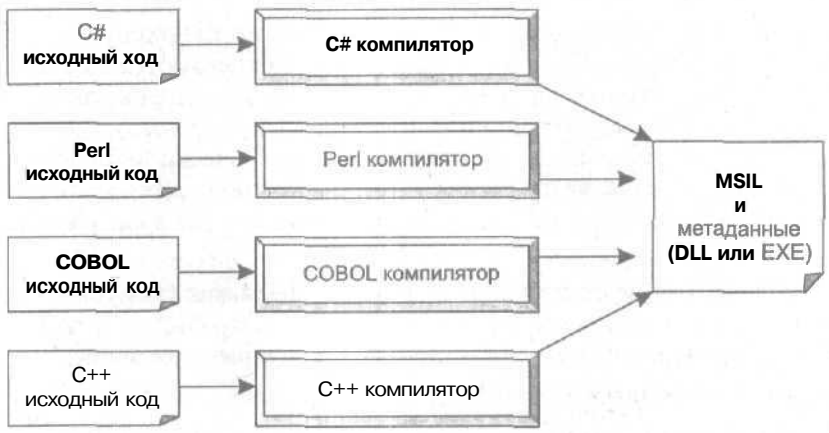


Рис. 1.2. Все компиляторы, ориентированные на .NET, генерируют IL-инструкции и метаданные

Может показаться смешным, но двоичные файлы .NET, для которых используются стандартные расширения DLL и EXE, по своему внутреннему содержанию не имеют абсолютно ничего общего с обычными исполняемыми файлами. Например, файлы DLL не предоставляют свои методы в распоряжение приложений на компьютере. В отличие от компонентов COM двоичные файлы .NET не описываются с помощью кода IDL и регистрируются в системном реестре. Однако, пожалуй, самое важное отличие заключается в том, что двоичные файлы .NET не содержат зависящих от платформы команд. Содержимое двоичных файлов .NET — это платформенно-независимый «промежуточный язык», который официально называется Microsoft Intermediate Language (MSIL, промежуточный язык Microsoft), или просто IL.

Обзор двоичных файлов .NET («сборки»)

Когда с помощью компилятора для платформы .NET создается модуль DLL или EXE, содержимое этого модуля — это так называемая сборка (assembly) на языке IL. Мы будем рассматривать сборки более подробно в главе 6. Однако для того, чтобы лучше понять особенности среды выполнения .NET, нам потребуется охарактеризовать хотя бы некоторые базовые свойства этого нового формата исполняемых файлов.

Как уже говорилось, сборка содержит код на «промежуточном языке» — IL. Назначение IL концептуально аналогично байт-коду Java — он компилируется в платформенно-специфичные инструкции, только когда это абсолютно необходимо. «Абсолютная необходимость» возникает тогда, когда к блоку инструкций IL (например, реализации метода) обращается для использования среда выполнения .NET.

Помимо инструкций IL, двоичные модули .NET содержат также метаданные, которые подробно описывают все типы, использованные в модуле. Например, если у вас внутри сборки есть класс Foo, то в метаданных этой сборки будет содержаться информация о базовом классе для Foo, какие интерфейсы предусмотрены для Fc o (если они вообще предусмотрены), а также полное описание всех методов, свойств и событий этого класса.

Во многих отношениях метаданные .NET являются значительным усовершенствованием по сравнению с классической информацией о типах в COM. Классические двоичные файлы COM обычно описываются с помощью ассоциированной библиотеки типов (которая очень похожа на двоичную версию кода IDL). Проблема с информацией о типах в COM заключается в том, что никто не может гарантировать вам, что эта информация окажется полной. Код IDL не может создать полный каталог внешних серверов, которые могут быть необходимы для нормальной работы содержащихся в модуле COM классов. Существование метаданных .NET, напротив, обеспечивается тем, что метаданные автоматически генерируются компилятором, создающим приложение .NET.

Метаданные описывают не только типы, используемые в сборке, но и саму сборку. Эта часть метаданных называется манифестом (manifest). В манифесте содержится информация о текущей версии сборки, об использованных ограничениях по безопасности, о поддерживаемом естественном языке (английском, русском и т. д.), а также список всех внешних сборок, которые потребуются для нормального выполнения. Нам предстоит рассмотреть в этой главе различные средства, которые можно использовать для анализа кода IL внутри сборки, метаданных для типов и манифеста.

Сборки из одного и нескольких файлов

В подавляющем большинстве случаев двоичный файл .NET и сборка — это одно и то же и между ними существует отношение «один-к-одному». Если мы будем говорить о создании .NET DLL, то понятия «двоичный файл» и «сборка» мы будем использовать как синонимы. Однако (подробнее об этом — в главе 6) такой подход верен не всегда. Сборка может состоять как из одного, так и из нескольких двоичных файлов. В сборке из одного файла (single file assembly) этот единственный файл содержит и манифест, и метаданные, и инструкции IL.

В сборке из нескольких двоичных файлов (multifile assembly) каждый двоичный файл называется модулем (module). При создании таких многофайловых сборок один из двоичных файлов должен содержать манифест сборки (в нем могут также находиться и другие данные, в том числе инструкции IL). Все остальные модули могут содержать только метаданные типов и инструкции IL.

Зачем может потребоваться создание многофайловой сборки? Единственная причина для этого — большая гибкость при развертывании приложения. Напри-

мер, если пользователь обращается к удаленной сборке, которая должна быть загружена на его локальный компьютер, среда выполнения загрузит лишь те модули сборки, которые действительно необходимы. Такое решение позволит избежать ненужного сетевого трафика и увеличить скорость работы программы.

Роль Microsoft Intermediate Language

Теперь, когда вы уже имеете представление о сборках .NET, мы можем рассмотреть Microsoft Intermediate Language (IL) — промежуточный язык Microsoft, более подробно. Код IL не зависит от платформы, на которой будет производиться выполнение. При этом компилятор для платформы .NET сгенерирует код IL вне зависимости от того, какой язык программирования (C#, Visual Basic.NET, Eiffel и т. п.) вы использовали для создания программы. Пожалуй, есть смысл продемонстрировать это более наглядно. В качестве примера мы создадим не самый сложный калькулятор. Единственное, что он у нас будет уметь делать — производить сложение 10 и 84. Ниже приведен код этого калькулятора на C#. Пока можно не задумываться об особенностях синтаксиса в этом примере, но отметьте для себя код, относящийся к методу Add().

```
// С пространствами имен мы познакомимся чуть ниже в этой главе
namespace Calculator
{
        using System;

        // В классе Calculator определен метод Add(), а также точка входа
        // приложения - метод Main()
        public class Calc
        {
                // Конструктор по умолчанию
                public Calc(){}

                public int Add(int x, int y)
                {
                        return x + y;
                }

                public static int Main(string[] args)
                {
                        // Создаем объект Calc и складываем два числа
                        Calc c = new Calc();
                        int ans = c.Add(10, 84);
                        Console.WriteLine("10 + 84 is {0}.", ans);
                        return 0;
                }
        }
}
```

После того как этот исходный файл будет обработан компилятором C# (csc.exe), в нашем распоряжении окажется исполняемый файл C# — сборка из одного файла. Внутри этого файла можно будет обнаружить манифест, инструкции IL и метаданные, описывающие класс Calc. Если мы заглянем внутрь этой сборки (с помощью чего — об этом мы скажем ближе к концу этой главы), то помимо всего прочего мы сможем найти следующий блок инструкций IL, относящихся к методу Add():

```
.method public hidebysig instance int32 Add(int32 x, int32 y) il managed
{
        // Размер кода 8 (0x8)
        .maxstack 2
        .locals ([0] int32 V_0)
        IL_0000: ldarg.1
        IL_0001: ldarg.2
        IL_0002:add
        IL_0003: stloc.0
        IL_0004: br.s IL_0006
        IL_0006: ldloc.0
        IL_0007: ret
} // Конец кода IL для метода Calc:Add()
```

Если большая часть строк в коде IL осталась для вас загадкой, не волнуйтесь. Код IL будет рассмотрен более подробно в главе 7. Пока самое важное — отметить, что компилятор C# генерирует не платформенно-зависимый набор инструкций, а код IL. То же самое справедливо и для других компиляторов .NET. Давайте создадим наш калькулятор на языке Visual Basic.NET:

```
' Калькулятор VB.NET
Module Module1
        ' Опять-таки, в классе Calc определен нетод Add()
        ' и точка входа для приложения
        Class Calc

                Public Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
                        ' Да! Теперь Visual Basic поддерживает ключевое слово 'return'
                        Return x + y
                End function
        End Class
        Sub Main()
                Dim ans As Integer
                Dim c As New Calc()
                ans = c.Add(10, 84)
                Console.WriteLine("10 + 84 is {0}.", ans)
        End Sub
End Module
```

Если мы поищем внутри сборки код, относящийся к методу Add(), то мы сможем обнаружить следующее:

```
.method public instance int32 Add(int32 x, int32 y) il managed
{
        // Размер кода 11 (0xb)
        .maxstack 2
        .locals init ([0] int32 Add)
        IL_0000: nop
        IL_0001: ldarg.1
        IL_0002: ldarg.2
        IL_0003: add.ovf
        IL_0004: stloc.0
        IL_0005:nop
        IL_0006: br.s IL_0008
        IL_0008:nop
        IL_0009: ldloc.0
        IL_000a: ret
1 // Конец метода Module1$Calc::Add
```

Как мы видим, получившийся код IL практически идентичен. Незначительные отличия возникают вследствие особенностей компиляторов C# и Visual Basic.NET.

Код приложений CSharpCalculator и VBCalculator можно найти в подкаталоге Chapter 1.

Преимущества IL

Возможно, к этому времени у вас уже созрел вопрос — а в чем, собственно, вообще могут состоять преимущества IL перед обычным набором платформенно-зависимых инструкций? Одно из преимуществ, про которое мы уже говорили, — возможность полного межъязыкового взаимодействия. Поскольку любой код на любом языке программирования .NET компилируется в стандартный набор инструкций IL, проблем во взаимодействии между блоками кода IL не будет. При этом взаимодействие будет производиться, как и положено, на двоичном уровне.

Еще одно возможное преимущество — потенциальная независимость от компьютерной платформы. Существует большая вероятность, что среда выполнения .NET будет распространена на самые разные компьютерные платформы и операционные системы (отличные от Windows). В результате .NET может пойти по стопам Java — то есть с помощью языков .NET можно будет создавать программы, которые будут работать под самыми разными операционными системами (и при этом в отличие от Java еще и пользоваться преимуществами языковой независимости!) Таким образом, .NET потенциально позволяет создавать приложения на *любом* языке, которые будут работать на *любой* платформе и под *любой* операционной системой.

Однако в отношении межплатформенности пока ключевое слово — «потенциально». На момент создания этой книги Microsoft официально не произнесла ни слова относительно возможности портирования среды выполнения .NET под другие операционные системы. Поэтому пока мы будем считать, что приложения .NET работают только под Windows.

Роль метаданных

Программистам, работающим с COM, хорошо знакома концепция Interface Definition Language (IDL, языка определения интерфейсов). IDL — это «метаязык», который позволяет, исключив любую двусмысленность, описать типы, используемые внутри сервера COM. IDL компилируется в двоичный формат (называемый библиотекой типов) с использованием компилятора midl.exe. Этот компилятор может использоваться любым языком, предназначенным для работы с COM.

IDL полностью описывает все типы данных, используемые в двоичном файле COM, но информация о самом этом двоичном файле в нем минимальна. Фактически она ограничивается номером версии (к примеру, 1.0, 2.0 или 2.4) и информацией о локализации (например, English, German, Russian). Кроме того, наличие или отсутствие метаданных (и их полноту) должен вручную контролировать создающий сервер COM программист — таким образом, необходимых метаданных в двоичном файле COM может вообще не оказаться.

В отличие от COM при использовании платформы .NET вам вообще не придется думать об IDL. Однако общий принцип описания типов в строго определенном двоичном формате остался.

Сборки .NET всегда содержат полные и точные метаданные, поскольку метаданные в них генерируются автоматически. Как и в IDL, в метаданных .NET содержится исчерпывающая информация об абсолютно всех типах, которые используются в сборке (классах, структурах, перечислениях и прочем), а также о каждом свойстве, методе или событии каждого типа.

Еще одно отличие метаданных .NET от информации IDL заключается в том, что метаданные .NET гораздо более подробны. В них перечислены все ссылки на внешние модули и сборки, которые потребуются для нормального выполнения сборки .NET. За счет этого можно считать сборки .NET фактически самодокументируемыми. В результате, к примеру, отпадает необходимость регистрировать двоичные файлы .NET в системном реестре (подробнее об этом будет сказано ниже).

Простой пример метаданных

Вот пример метаданных для метода Add() нашего приложения CSharpCalculator (метаданные для этого метода в VBCalculator будут точно такими же):

```
Method #2
-------------------------------------------------------
MethodName      : Add (06000002)
Flags           : [Public] [HideBySig] [ReuseSlot] (00000086)
RVA             : 0x00002058
ImplFlags       : [IL] [Managed] (00000000)
CalcCnvntn      : [DEFAULT]
hasThis
ReturnType      : I4
2 Arguments
        Argument #1: I4
        Argument #2: I4
2 Parameters
        (1) ParamToken : (08000001) Name : x flags: [none] (00000000) default:
        (2) ParamToken : (08000002) Name : y flags: [none] (00000000) default:
```

В коде ясно представлены название метода, тип возвращаемого значения и данные об ожидаемых аргументах. Как мы помним, вручную никаких метаданных мы не писали — за нас все сделал компилятор C#.

Кто будет обращаться к метаданным? И сама среда выполнения .NET (очень часто), и самые разные средства разработки и отладки. Например, средство IntelliSense в Visual Studio.Net (которое пытается помочь вам закончить начатую строку) берет необходимую ему информацию именно из метаданных. Метаданные активно используются утилитами просмотра, отладки и, конечно, самим компилятором C#.

Компиляция IL в платформенно-зависимые инструкции

Поскольку в сборках, как мы выяснили, содержится платформенно-независимый код IL, а выполняются в конечном итоге именно платформенно-зависимые инструкции, кто-то должен взять на себя работу по компиляции IL в такие инструкции. Этот «кто-то» называется "just-in-time compiler" (JIT) — компилятор времени выполнения. JIT часто ласково называют "jitter" (дрожание, трепет). JIT для

перевода IL в платформенно-зависимые инструкции входит в состав среды выполнения .NET. Используя код IL, разработчики могут не думать об особенностях архитектуры CPU данного компьютера — эти особенности будут учтены JIT.

Откомпилированные из IL платформенно-зависимые инструкции JIT помещает в кэш-памяти, что очень сильно ускоряет работу приложения. Предположим, был вызван метод `Bar()` класса Foo. При первом вызове этого метода JIT откомпилирует относящийся к этому методу код IL в платформенно-зависимые инструкции. При повторных вызовах этого метода JIT уже не будет заниматься компиляцией, а просто возьмет уже готовый откомпилированный код из кэша в оперативной памяти.

Типы .NET и пространства имен .NET

Сборка (не важно, однофайловая или многофайловая) может содержать любое количество самых разных типов. В мире .NET *тип* — это общий термин, который может относиться к классам, структурам, интерфейсам, перечислениям и прочему. При создании приложения .NET (например, на языке C#) вам потребуется организовывать взаимодействие этих типов. Например, сборка может определять класс с несколькими интерфейсами; один интерфейс может принимать в качестве параметров только значения определенного перечисления.

У вас есть возможность использовать пространства имен при создании ваших собственных типов. Пространство имен — это логическая структура для организации имен, используемых в приложении .NET. Основное назначение пространств имен — предупредить возможные конфликты между именами в разных сборках.

Вот пример: вы создаете приложение типа Windows Forms, которое обращается к двум внешним сборкам. В каждой сборке есть тип с именем GoCart, при этом эти типы отличаются друг от друга. При написании кода вы можете точно указать, к какому именно типу и из какой сборки вы обращаетесь. Для этого достаточно к имени типа добавить имя соответствующего пространства имен: например, `CustomVehicals.GoCart` или `SlowVehicals.GoCart`. Более подробно мы разберем применение пространств имен ниже в этой главе.

Основы Common Language Runtime - - среды выполнения .NET

После того как мы познакомились с типами, сборками, метаданными и IL, настало время рассмотреть среду выполнения .NET — CLR более подробно. Среду выполнения (runtime) можно рассматривать как набор служб, необходимых для работы блока программного кода. К таким службам можно отнести и требуемые библиотеки. Например, если вы создали приложение MFC, то в качестве компонента среды выполнения вам потребуется весьма объемистая библиотека времени выполнения Microsoft Foundation Classes — mfc42.dll Программы на Visual Basic привязаны к такому компоненту среды выполнения, как библиотека `msvbvm60.dll`, а программам на Java необходим большой набор файлов, входящих в состав виртуальной машины Java.

Своя среда выполнения требуется и приложениям .NET. Главное отличие этой среды выполнения от всех тех, которые были перечислены выше, заключается в том, что единая среда выполнения .NET используется приложениями, написанными на любых языках программирования .NET. Как уже говорилось выше, среда выполнения .NET носит официальное название Common Language Runtime (CLR).

Сама CLR состоит из двух главных компонентов. Первый компонент — это ядро среды выполнения, которое реализовано в виде библиотеки mscoree.dll. При обращении к приложению .NET mscoree.dll автоматически загружается в память, и, в свою очередь, эта библиотека управляет процессом загрузки в память сборки данного приложения. Ядро среды выполнения ответственно за множество задач. Оно занимается поиском физического местонахождения сборки, обнаружением внутри сборки запрошенного типа (класса, интерфейса, структуры и т. п.) на основе информации метаданных, компилирует IL в платформенно-зависимые инструкции, выполняет проверки, связанные с обеспечением безопасности, — и этот перечень далеко не полон.

Второй главный компонент CLR — это библиотека базовых классов. Сама библиотека разбита на множество отдельных сборок, однако главная сборка библиотеки базовых классов представлена файлом mscorlib.dll. В библиотеке базовых классов содержится огромное количество типов для решения распространенных задач при создании приложения. Приложение .NET будет обязательно использовать сборку mscorlib.dll и по мере необходимости — другие сборки (как встроенные, так и создаваемые вами самими).

На рис. 1.3 представлен путь, который проходит исходный код приложения, прежде чем воплотиться в выполнение каких-либо действий на компьютере.

Стандартная система типов CTS

Мы уже говорили, что стандартная система типов (Common Type System, CTS) — это формальная спецификация, которая определяет, как какой-либо тип (класс, структура, интерфейс, встроенный тип данных и т. п.) должен быть определен для его правильного восприятия средой выполнения .NET. CTS определяет синтаксические конструкции (в качестве примера можно взять перегрузку операторов), которые могут поддерживаться, а могут и не поддерживаться конкретным языком программирования .NET. Если вы хотите создавать сборки, которые смогут использоваться всеми языками программирования .NET, вам придется при создании типов следовать правилам Common Language Specification — CLS. А сейчас мы рассмотрим особенности тех типов, к которым применяется спецификация CTS.

Классы CTS

Концепция классов — краеугольный камень любого объектно-ориентированною программирования. Она поддерживается всеми языками программирования .NET. Класс (class) — это набор свойств, методов и событий, объединенных в единое целое. Как, наверное, вы и предполагали, в CTS предусмотрены абстрактные члены классов, что обеспечивает возможность применения полиморфизма в производных классах. Множественное наследование в CTS запрещено. Самые важные характеристики классов представлены в табл. 1.1.

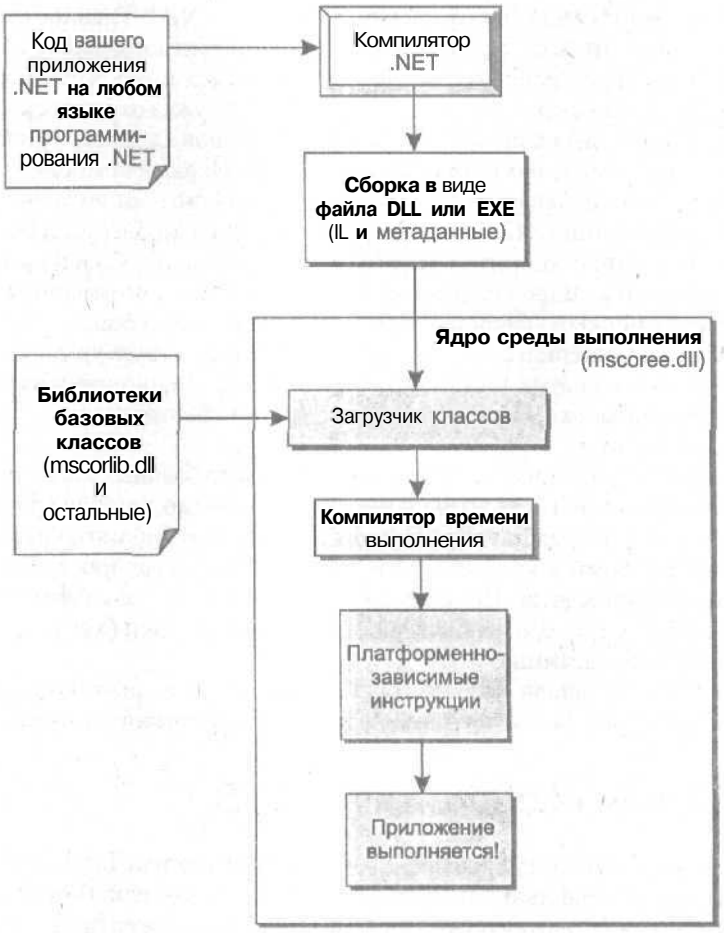


Рис. 1.3, Роль среды выполнения .NET

Таблица 1.1. Самые важные характеристики классов CTS

Характеристика	Ее смысл
Является ли класс «закрытым»?	Закрытые классы не могут становиться базовыми для других классов
Предусмотрены ли в классе какие-либо интерфейсы?	Интерфейс — это набор абстрактных членов, который обеспечивает связь между объектом и пользователем. В CTS в классе может быть любое количество интерфейсов
Является ли класс абстрактным?	Объекты абстрактных классов создать невозможно. Единственное назначение абстрактных классов — выполнять роль базовых для других классов. За счет механизма наследования абстрактные классы обеспечивают производные классы общими наборами членов
Какова область видимости для данного класса?	Для каждого класса должен быть определен атрибут области видимости (visibility). Как правило, значение этого атрибута определяет, можно ли обращаться к этому классу из внешних сборок или только из той, которая его содержит

Структуры CTS

Помимо классов в CTS также предусмотрена концепция структур (structures). Если; вы работали с C, возможно, вы удивитесь, что этот пользовательский тип данных сохранился и в мире .NET (правда, надо отметить, что он немного изменился). В принципе, структуры можно грубо рассматривать как упрощенные разновидности классов (подробнее о различиях между классами и структурами будет рассказано в главе 2). Структуры CTS могут иметь любое количество конструкторов с параметрами (конструктор без параметров зарезервирован). С помощью конструкторов с параметрами вы можете установить значение любого поля объекта структуры в момент создания этого объекта. Например:

```
// Определяем структуру C#
struct Baby
{
        // В структуре могут быть поля:
        public string name;
        // В структуре можно определить конструкторы (но только с параметрами):
        public Baby (string name)
        {this.name = name; }

        // В структурах могут быть определены методы:
        public void Cry()
        { Console.WriteLine ("Waaaaaaaaaah!!!"); }

        public bool IsSpleepingC) { return false; }
        public bool IsChanged() { return false; }
}
```

А вот наша структура в действии:

```
// Добро пожаловать в мир малютки Макса!

Baby barnaBaby = new Baby ("Max");
Console.WriteLine ("Changed?: {0}", barnaBaby.IsChanged().ToString());

Console.WriteLine ("Sleeping?: {0}", barnaBaby.IsSleeping().ToString());

// А теперь Макс покажет нам свою подлинную сущность:
for{int i=0; i<10000; 1++)
        barnaBaby.Cry();
```

Все CTS-совместимые структуры произведены от единого базового класса System.ValueType. Этот базовый класс определяет структуру как тип данных для работы только со значениями, но не с ссылками. В структуре может быть любое количество интерфейсов. Однако структуры не могут быть унаследованы от остальных типов данных и они всегда являются «закрытыми» — то есть они не могут выступать в качестве базовых для целей наследования.

Интерфейсы CTS

Интерфейсы (interfaces) — это просто наборы абстрактных методов, свойств и определений событий. В отличие от классической технологии COM, интерфейсы .NET не являются производными от единого общего интерфейса, каким в мире COM был интерфейс IUnknown. В интерфейсах самих по себе смысла не очень много. Однако если мы знаем, что какой-либо класс реализует известный нам интерфейс,

мы вправе требовать от этого класса определенной функциональности. При создании своего собственного интерфейса на .NET-совместимом языке программирования вы можете произвести этот интерфейс сразу от нескольких базовых интерфейсов. Подробнее про использование интерфейсов в программах на C# будет рассказано в главе 6.

Члены типов CTS

Как мы уже говорили, в классах и структурах может быть любое количество членов. Член (member) — это либо метод, либо свойство, либо поле, либо событие. Подробнее про члены типов в мире .NET будет рассказано в нескольких ближайших главах. Однако нам сейчас важно отметить, что для любого члена в .NET существует набор характеристик.

Например, любой член в .NET характеризуется своей областью видимости (public, private, protected и т. д.). Член можно объявить как абстрактный, чтобы воспользоваться возможностями полиморфизма при работе с производными классами. Члены могут быть статическими (static, такие члены могут совместно использоваться всеми объектами данного класса) и обычными — принадлежащими только одному объекту данного класса.

Перечисления CTS

Перечисление (enumeration) — это удобная программная конструкция, которая позволяет вам объединять пары имя — значение под указанным вами именем перечисления. Предположим, что вы создаете компьютерную игрушку, в которой играющий сможет выбирать из трех типов персонажей — волшебника (Wizard), воина (Fighter) или вора (Thief). В этой ситуации очень удобно будет воспользоваться перечислением:

```
// Перечисление C#:
enum PlayerType
{ Wizard=100, Fighter=200, Thief=300 };
```

В CTS все перечисления являются производными от единственного базового класса System.Enum. Как мы убедимся в будущем, этот базовый класс содержит множество полезных членов, которые помогут нам в извлечении (и выполнении прочих операций) с парами имя — значение.

Делегаты CTS

Делегаты (delegates) — в мире .NET это безопасный для типов эквивалент указателя на функцию в C. Однако между ними есть и существенное отличие. Делегат .NET — это уже не просто адрес в оперативной памяти, а класс, производный от базового класса MulticastDelegate. Делегаты очень полезны в тех ситуациях, когда вам нужно, чтобы одна сущность передала вызов другой сущности. Делегаты — это краеугольный камень в технологии обработки событий .NET (об этом — в главе 5).

Встроенные типы данных CTS

Конечно же, в .NET предусмотрен богатый набор встроенных типов данных. Помимо всего прочего, этот набор еще и един для всех языков программирования

.NET. Названия типов данных в языках .NET могут выглядеть по-разному, но эти названия — всего лишь псевдонимы для встроенных системных типов данных .NET, определенных в библиотеке базовых типов. Перечень встроенных типов данных .NET представлен в табл. 1.2,

Таблица 1.2. Встроенные типы данных CTS

Встроенный тип данных .NET	Название в Visual Basic.NET	Название в C#	Название в Managed C++
System. Byte	Byte	byte	char
System.SByte	Не поддерживается	sbyte	signed char
System. Int16	Short	short	short
System. Int32	Integer	int	int или long
System. Int64	Long	long	_int64
System. UInt16	Не поддерживается	ushort	unsigned short
System. UInt32	Не поддерживается	uint	unsigned int или unsigned long
System.UInt64	Не поддерживается	ulong	unsigned _int64
System.Single	Single	float	float
System. Double	Double	double	double
System.Object	Object	object	Object*
System. Char	Char	char	_wchar_t
System.String	String	string	String*
System. Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	bool

Как видно из таблицы, не все языки .NET могут работать с некоторыми встроенными типами данных CTS. Поэтому очень важно определить такой набор типов (и программных конструкций), с которым гарантированно смогут работать любые .NET-совместимые языки программирования. Такой набор есть, и он называется CLS.

Основы CLS

Нет необходимости доказывать, что одни и те же программные конструкции в разных языках выглядят абсолютно по-разному. Например, в C# объединение строк (конкатенация) производится с помощью оператора плюс (+), в то время как в Visual Basic для этой же цели используется амперсанд (&). А вот как выглядит в разных языках функция, не возвращающая значений:

```
' Функция (подпроцедура) VBasic. которая ничего не
' возвращает (возвращает значение типа void):
Public Sub Foo()
       ' Что-то делаем...
End Sub

// Такая же функция в C#:
public void FooO
```

```
{
        // Делаем то же самое...
}
```

Как мы уже видели, для среды выполнения .NET такая разница в синтаксисе абсолютно безразлична: все равно соответствующие компиляторы (в нашем случае vbc.exe и csc.exe) создадут одинаковый код IL. Однако языки программирования отличаются не только синтаксисом, но и возможностями. Например, в одних языках программирования разрешена перегрузка операторов, а в других — нет. Одни языки могут использовать беззнаковые (unsigned) типы данных, а в других такие типы данных не предусмотрены. Вывод очевиден — нам нужны некие единые правила для всех языков .NET. Если мы им следуем, то гарантируется, что программные модули, написанные на разных языках, будут нормально взаимодействовать друг с другом. Такой набор правил определен в спецификации CLS (Common Language Specification).

Набор правил, определяемый CLS, не только гарантирует нормальное взаимодействие блоков кода, созданных на разных языках. Такой набор правил еще и определяет минимальные требования, которые предъявляются к любому .NET-совместимому компилятору. Необходимо помнить, что в CLS — это лишь часть тех возможностей, которые определены в CTS.

Правилам CLS должны удовлетворять и инструментальные средства среды разработки — если мы хотим обеспечить межъязыковое взаимодействие, они должны генерировать только такой код, который соответствует требованиям CLS. У каждого правила CLS есть простое название (например, CLS Rule 6 — правило CLS номер 6). Бот пример одного из правил (это самое важное правило — правило номер 1):

- Правило 1. Правила CLS относятся только к тем частям типа, которые предназначены для взаимодействия за пределами сборки, в которой они определены.

Из этого правила явствует, что во внутренней логике при реализации какого-либо типа вы можете сколько угодно нарушать правила CLS — это ни на что не повлияет. Например, предположим, что вы создаете приложение .NET, которое взаимодействует с внешним миром с помощью трех классов, а в каждом из этих классов есть только одна функция. Правилам CLS в этом случае должны удовлетворять только три этих функции-члена (в отношении области видимости, соглашений об именовании, типов принимаемых параметров и т. д.). Во внутренней реализации этих функций, классов или приложения в целом может быть сколько угодно отступлений от правил CLS — за пределами вашего программного модуля никто об этом никогда не узнает.

Конечно, в CLS существует не только правило 1, но и множество других правил. В CLS, к примеру, строгие требования предъявлены к представлению символьных значений, к определению перечислений, к использованию статических членов и т. д. Однако заучивать эти правила совершенно не обязательно (конечно, если вы не заняты созданием .NET-совместимого компилятора для своего собственного языка программирования). Если вам потребовалась дополнительная информация по CLS, произведите поиск в MSDN по словосочетанию «Collected CLS Rules».

Работа с пространствами имен

Мы закончили обзор среды выполнения .NET и теперь обратимся к особенностям библиотеки базовых классов .NET. Важность библиотек кода очевидна. Например, библиотека MFC определяет набор классов C++ для создания диалоговых окон, меню и панелей управления. В результате программисты могут не заниматься изобретением того, что давным-давно уже сделано до них, а сосредоточиться на уникальных аспектах создаваемого ими приложения. Аналогичные средства существуют и в Visual Basic, и в Java, и во всех остальных языках программирования.

Однако в отличие от MFC, Visual Basic и Java в C# не существует библиотеки базовых классов *только для этого языка*. Можно сказать, что библиотека базовых классов C# вообще не существует. Вместо этого разработчики на C# используют библиотеку базовых типов *для всей среды .NET*. А для лучшей организации типов внутри этой библиотеки используется концепция пространств имен.

Главное отличие от библиотек, привязанных к конкретному языку (типа MFC), заключается в том, что в любом .NET-совместимом языке используются те же самые типы и те же самые пространства имен, что и в C#. Вот три приложения (весьма напоминающих классическое Hello, World) на трех разных .NET-совместимых языках: C#, VB.NET и Managed C++ (MC++).

```
// Привет от C#
using System;
public class MyApp
{
        public static void Main()
        {
                Console.WriteLine ("Hi from C#");
        }
}
```

```
' Привет от VB.NET
Imports System
Public Module MyApp

        Sub MainO
                Console.WriteLine ("Hi from VB");
        End Sub
End Module
```

```
// Привет от Managed C++
using namespace System:
// Обратите внимание! Среда выполнения .NET в C++ сама собой поиещает глобальную
// функцию main внутрь определения класса
void main()
{
        Console::WriteLine("Hi from MC++");
}
```

Обратите внимание, что если нужен класс Console, то в любом языке .NET используется одно и то же пространство имен System. Если не обращать внимания на синтаксические различия, то код приложений на разных языках .NET практически идентичен. Платформе .NET свойственно изящество единого стиля программирования.

Важнейшие пространства имен .NET

Эффективность работы программиста, использующего .NET, напрямую зависит от того, насколько он знаком с тем массивом типов, который определен в пространствах имен библиотеки базовых классов. Самое важное пространство имен в C# - это System. В нем определены классы, которые обеспечивают самые важные функции C#. Вам не удастся создать ни одно работоспособное приложение C# без использования этого пространства имен.

Пространство имен — это просто способ организации типов (классов, перечислений, интерфейсов, делегатов и структур) в единую группу. Конечно, обычно в одном пространстве имен объединяются взаимосвязанные типы. Например, тип System.Drawing содержит набор типов, которые призваны помочь вам в организации вывода изображений на графическое устройство. В .NET предусмотрены пространства имен для организации типов для работы с базами данными, Web, многопоточностью, защитой данных и множества других задач. В табл. 1.3 приведены некоторые (далеко не все) пространства имен .NET.

Таблица 1.3. Пример пространства имен .NET

Пространство имен .NET	Назначение
System	Внутри — множество низкоуровневых классов для работы с простыми типами, выполнения математических операций, сборки мусора и т. п.
System.Collections	Для работы с контейнерными объектами, такими как ArrayList, Queue, SortedList
System.Data System.Data.Common System.Data.OleDb System.Data.SqlClient	Для обращений к базам данных. В книге этой теме посвящена специальная глава
System.Diagnostics	В этом пространстве имен содержатся многочисленные типы, используемые .NET-совместимыми языками для трассировки и отладки программного кода
System.Drawing System.Drawing.Drawing2D System.Drawing.Printing	Типы для примитивов GDI+ — растровых изображений, шрифтов, значков, поддержки печати. Предусмотрены также специальные классы для вывода более сложных изображений
System.IO	Как следует из названия, в этом пространстве имен объединены типы, отвечающие за операции ввода-вывода — в файл, буфер и т.п.
System.Net	Это пространство имен (как и все остальные, связанные с ним) содержит типы, относящиеся к передаче данных по сети (запрос — ответ, создание сокетов и т. п.)
System.Reflection System.Reflection.Emit	Классы, предназначенные для обнаружения, создания и вызова во время выполнения пользовательских типов
System.Runtime.InteropServices System.Runtime.Remoting	Средства для взаимодействия с «традиционным» кодом (Win32 DLL, COM-серверы) и типы, используемые для удаленного доступа (например, по коммутируемым соединениям)
System.Security	В мире .NET средства обеспечения безопасности интегрированы как со средой выполнения, так и с библиотекой базовых типов. В этом пространстве имен находятся классы для работы с разрешениями, криптографией и т. п.

Пространство имен .NET	Назначение
System.Threading	Скорее всего, вы уже угадали — это пространство имен для типов, которые используются при работе с потоками (например, Mutex, Thread или Timeout)
System.Web	Классы, которые предназначены для использования в web-приложениях, включая ASP.NET
System.Windows.Forms	Классы для работы с элементами интерфейса Windows — окнами, элементами управления и прочим
System.XML	Множество классов для работы с данными в формате XML

Использование пространств имен в коде приложения

Как мы помним, пространство имен — это средство для логической группировки типов. С человеческой точки зрения выражение System.Console означает тип Console в пространстве имен System. Однако с точки зрения среды выполнения .NET System.Console — это единая сущность, к которой можно обратиться разными способами.

Слово using нужно только вам — так проще будет обращаться к типам в конкретном пространстве имен. Если по каким-либо причинам использовать слово using вы не хотите, вполне можно обойтись и без него. Давайте рассмотрим это на примере. Предположим, что вы создаете обычное оконное приложение Windows, которое должно представлять на круговой диаграмме информацию, извлекаемую из базы данных. Кроме того, вам еще нужно поместить на главную форму своего приложения растровый рисунок с логотипом компании. Немного подумав, мы можем определить, что в нашем приложении нам потребуются классы из следующих пространств имен:

```
//Пространства имен для использования в нашем приложении
using System;                //Без главного пространства имен не обойтись
using System.Drawing;        //для вывода изображений
using System.Windows.Forms;  //для элементов интерфейса
using System.Data;           //для доступа к базе данных
using System.OleDb;          //если к базе данных мы обращаемся по OLE DB
```

После того как мы определим использование конкретного пространства имен (с использованием ключевого слова using), мы можем обращаться к типам, содержащимся в этом пространстве. Например, если нам потребовалось создать экземпляр класса Bitmap (определенном в пространстве имен System.Drawing), код может быть таким;

```
//Явно указываем использование пространства имен:
using System.Drawing;

class MyClass
{
        public void DoIt()
        {
                //Создаем растровое изображение 20 на 20 пикселов
                Bitmap bm = new Bitmap (20, 20);
                ...
        }
}
```

Поскольку мы явно указали использование пространства имен System.Drawing с помощью ключевого слова using, компилятор сможет понять, что класс Bitmap — это член данного пространства имен. Если в примере, приведенном выше, мы опустим строку со словом using, мы получим сообщение компилятора об ошибке. Однако можно обойтись и без using, если использовать полное имя класса, как в следующем примере:

```
//Обратите внимание - никаких указаний на пространства имен!
class MyClass
{
        public void DoIt()
        {
                //Используем полное имя
                System.Drawing.Bitmap bm = new System.Drawing.Bitmap (20. 20);
                ...
        }
}
```

Главная идея, я думаю, понятна; если вы явно указываете используемое пространство имен, строки при обращении к классам этого пространства имен получаются гораздо меньшего размера.

Обращения к внешним сборкам

Помимо того что вы можете явно указать используемое пространство имен с помощью ключевого слова using, иногда вам может потребоваться еще и явно указать физическое местонахождение сборки с необходимым кодом IL. Многие важнейшие пространства имен .NET физически связаны с файлом mscorlib.dll. Типы пространства имен System.Drawing физически «живут» внутри файла System.Drawing.dll. По умолчанию встроенные сборки .NET находятся в подкаталоге <имя_диска>:\WINNT\Microsoft.NET\Framework\<номер_версии>, как показано на рис. 1.4.

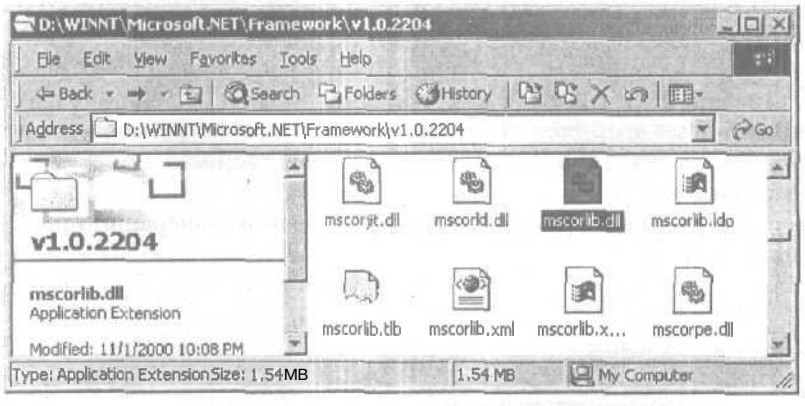


Рис. 1.4. Библиотеки базовых классов .NET

В зависимости от того, какие средства разработки вы используете для создания приложений .NET, существует много способов сообщить компилятору, какие имен-

но сборки вы собираетесь задействовать во время процесса компиляции. Про эти способы будет рассказано ниже.

Если вам стало немного не по себе от мысли о том, сколько информации о пространствах имен и о типах вам придется осваивать, помните, что помнить все типы всех пространств имен совершенно незачем. Если вы создаете консольное приложение, вам можно забыть о всех типах System.Windows.Forms и System.Drawing (а, скорее всего, и о многих других). Если же вы разрабатываете редактор изображений, вам вряд ли потребуются интерфейсы для доступа к базам данных. Типы пространств имен можно осваивать постепенно, по мере необходимости.

Как получить дополнительную информацию о пространствах имен и типах

Во всех главах этой книги мы будем осваивать различные возможности платформы .NET, используя пространства имен и содержащиеся в них типы. Книга вряд ли стала бы лучше, если бы мы рассмотрели в ней все без исключения типы во всех встроенных пространствах имен. Вы должны уметь находить информацию о нужных вам типах и пространствах имен самостоятельно, тем более что инструментов для этой цели много. Вот их краткий перечень:

- документация .NET SDK (в MSDN);
- утилита ILDasm.exe;
- web-приложение ClassView;
- графическое приложение WinCV;
- ObjectBrowser, входящий в комплект Visual Studio.NET.

Вряд ли вас нужно учить тому, как использовать MSDN (намекнем только, что внутри Visual Studio.NET можно попробовать начать с кнопки F1). А вот про все остальные утилиты стоит поговорить подробнее. Начнем с ILDasm.exe, ClassView и WinCV. Все эти утилиты поставляются вместе с .NET SDK.

ILDasm.exe

Официальное название ILDasm.exe звучит как Intermediate Language Disassembler utility (утилита дизассемблирования промежуточного языка). Эта утилита позволяет просмотреть содержимое любой сборки .NET (файла DLL или EXE) — ее манифест, метаданные типов и инструкции IL. При этом все операции производятся с использованием дружественного графического интерфейса. Просто запустите ILDasm.exe и через меню File ▸ Open откройте нужную сборку. В порядке демонстрации мы откроем сборку mscorlib.dll (рис. 1.5). Путь к открытой нами сборке будет показан в заголовке окна ILDasm.

Как мы видим, структура сборки представлена в самом обычном формате с деревом и узлами. Каждый метод, свойство, вложенный класс, как и все остальные типы, представлены специальными значками (в текстовом дампе дизассемблера эти значки будут заменены на аббревиатуры, состоящие из трех символов). Самые распространенные значки и соответствующие им аббревиатуры ILDasm приведены в табл. 1.4.

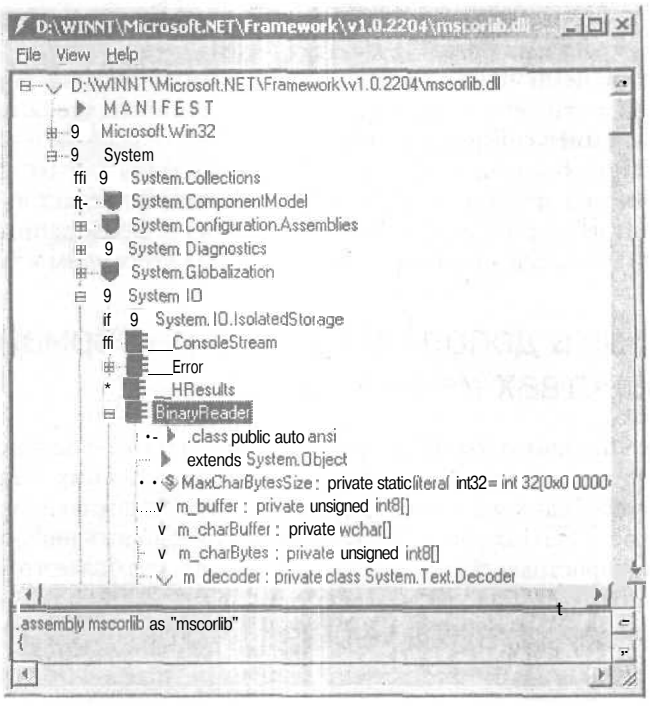


Рис. 1.5. ILDasm.exe — ваш лучший друг в мире .NET

Таблица 1.4. Условные обозначения в ILDasm

Значки ILDasm	Соответствующие аббревиатуры в текстовом дампе	Значение
	(.dot)	Показывает, что для типа может быть отображена дополнительная информация. В некоторых случаях двойной щелчок на этом значке позволяет перейти к связанному с ним узлу в дереве
	[NSP]	Пространство имен
	[CLS]	Класс. Вложенные классы представлены в формате <внешний_класс>$< внутренний_класс>
	[VCL]	Структура
	[INT]	Интерфейс
	[FLD]	Поле (то есть открытые данные), определенное некоторым типом
	[STF]	Статическое поле (то есть поле, которое принадлежит всем объектам данного класса)
	[MET]	Метод
	[STM]	Статический метод
	[PTY]	Свойство

Помимо просмотра информации о типах и их членах, ILDasm позволяет также получать информацию об инструкциях IL, относящихся к выбранному вами типу. В качестве примера можно найти и щелкнуть два раза мышью на значке конструктора по умолчанию для класса System.IO.BinaryWriter. Откроется отдельное окно, подобное представленному на рис. 1.6.

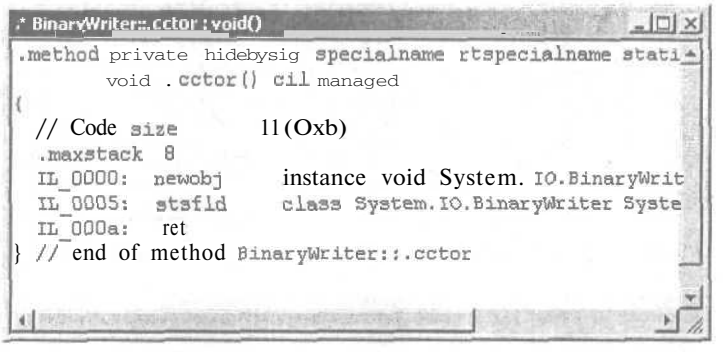


Рис. 1.6. Просмотр кода IL в ILDasm

Выгрузка в файл иерархии типов и членов сборки

ILDasm обладает замечательной возможностью выгружать иерархию исследуемой вами сборки в текстовый файл. Потом, к примеру, полученный дамп можно изучать в любимой кофейне (или пивной). Чтобы создать текстовый дамп, откройте нужную сборку, в меню File выберите команду Dump TreeView и укажите имя для создаваемого текстового файла. Обратите внимание, что графические значки будут заменены соответствующими текстовыми аббревиатурами, как это показано в табл. 1.4. Пример текстового дампа представлен на рис. 1.7.

Выгрузка в файл вместе с инструкциями IL

В файл можно выгружать не только типы и члены типов исследуемой вами сборки, но и относящийся к объектам сборки код IL. Для этого в ILDasm предназначена другая команда: File ▸ Dump. По умолчанию для дампов с инструкциями IL используется расширение *.il. На рис. 1.8 представлен код IL, относящийся к методу GetType() сборки mscorlib.dll (мы подробно рассмотрим этот метод в главе 7).

Просмотр метаданных типов

В ILDasm есть еще одна возможность, о которой обязательно следует упомянуть. С помощью ILDasm можно просматривать метаданные типов — ту информацию о типах сборки, которую генерирует .NET-совместимый компилятор для среды выполнения .NET. Для просмотра метаданных типов достаточно загрузить сборку в ILDasm и нажать клавиши Ctrl+M. Метаданные типов для приложения TestApp.exe (мы его вскоре создадим) представлены на рис. 1.9.

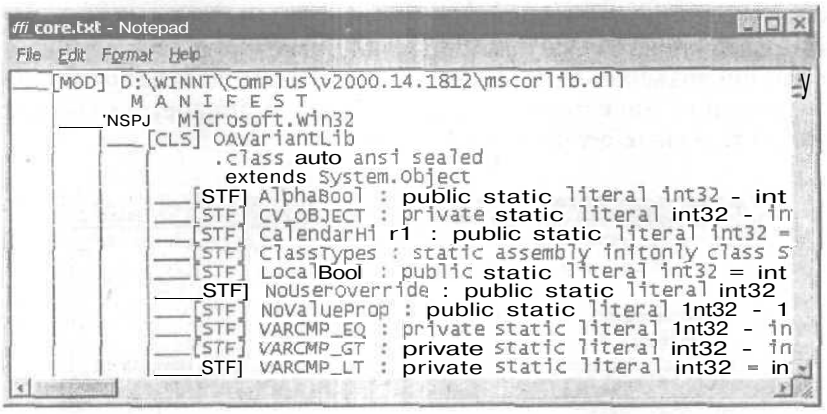


Рис. 1.7. Текстовый дамп иерархии сборки, созданный ILDasm

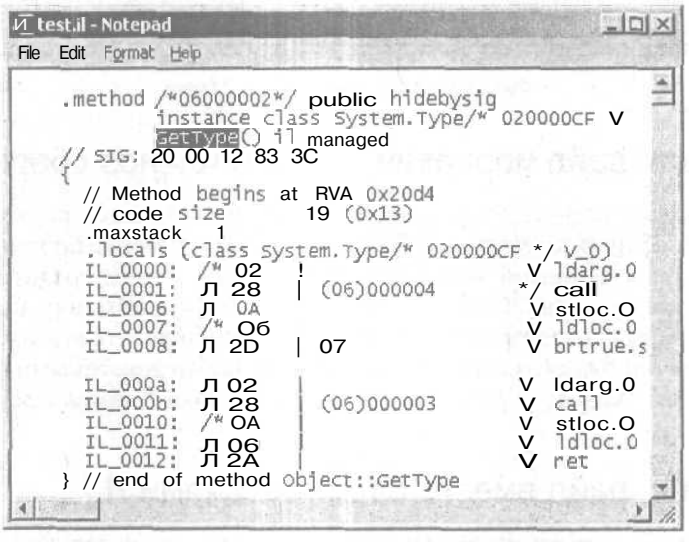


Рис. 1.8. Текстовый дамп сборки вместе с кодом IL

Если вы еще не заметили сами, скажем, что ILDasm очень похож на утилиту OLE/COM Object Viewer. Oleview — это средство для получения информации о серверах COM и изучения кода IDL, который содержится в двоичных файлах COM. ILDasm — это средство для просмотра иерархии типов сборок .NET, связанного с ними кода IL и метаданных типов.

Web-приложение ClassViewer

Заглянуть внутрь сборок .NET можно с помощью еще одного средства — приложения ClassViewer. Это приложение входит в состав примеров .NET SDK. Для его запуска вам достаточно установить примеры .NET SDK, а затем в Internet Explorer открыть страницу по адресу http://localhost/ClassViewer/Default.aspx. ClassViewer позволит отслеживать отношения типов внутри сборок, используя web-интерфейс (рис. 1.10).

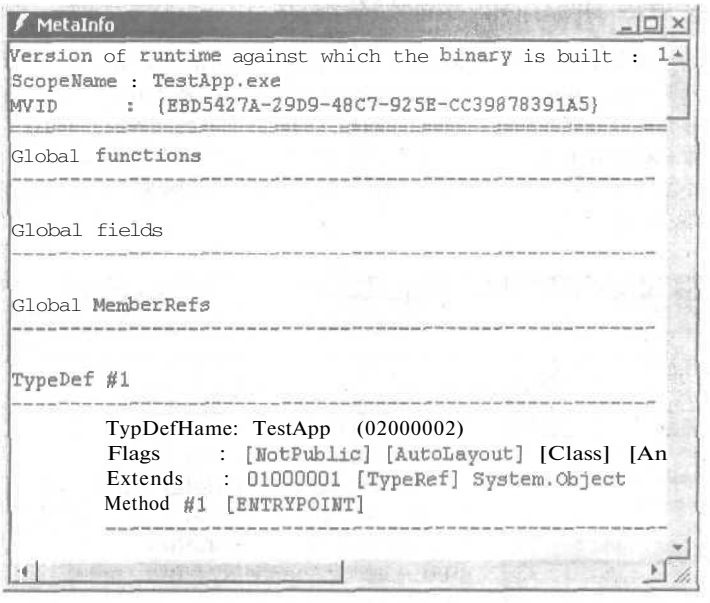


Рис. 1.Э. Просмотр метаданных типов в ILDasm

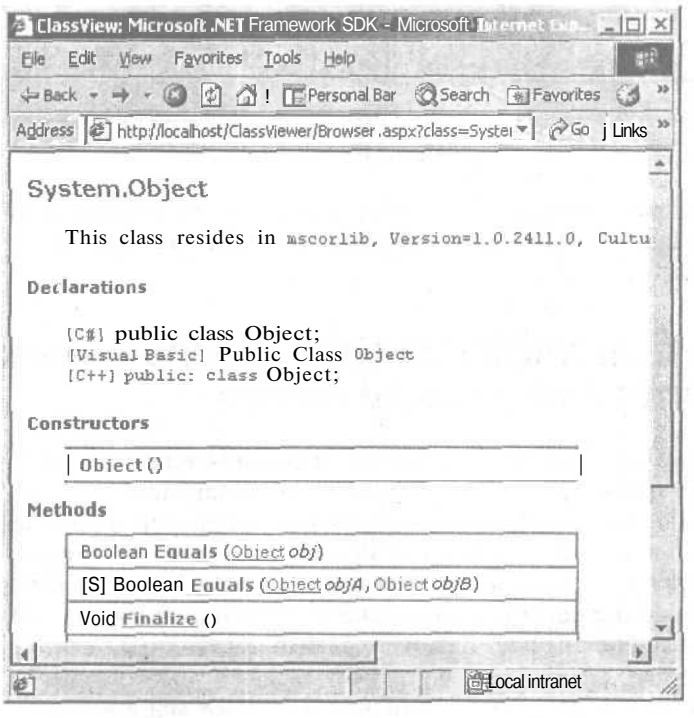


Рис. 1.10. Просмотр типов в web-приложении ClassViewer

Графическое приложение WinCV

Последнее *приложение*, с которым мы *познакомимся*, называется WinCV.exe (от Windows Class Viewer). Это приложение позволяет просматривать определения типов C# в библиотеках базовых типов. Интерфейс этого приложения очень прост: наберите имя интересующего вас типа в строке поиска, и в окне Selected Class будут показаны его члены. На рис. 1.11 представлены члены класса `System.Windos.Forms.ToolTip`.

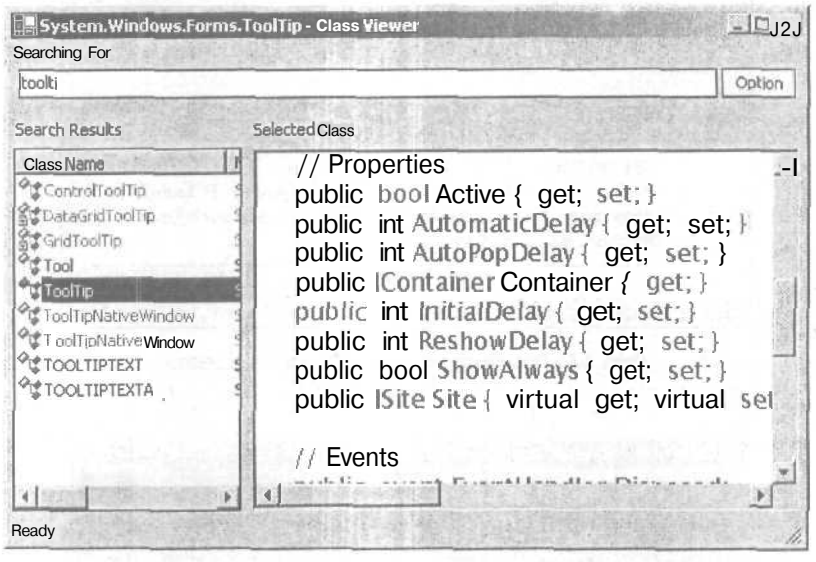


Рис. 1.11. Окно WinCV

Мы с вами рассмотрели приложения для просмотра сборок и типов C#. Следующая наша задача — познакомиться с теми средствами, которые используются для создания приложений C#.

Создание приложений C# с использованием компилятора командной строки

Создавать приложения C# можно с помощью компилятора командной строки csc.exe (C Sharp Compiler). Этот компилятор поставляется с .NET **SDK,** кроме того, его можно свободно загрузить с web-сайта Microsoft. В этом разделе мы используем этот компилятор для создания приложения на C#, которое будет называться TestApp.exe. Прежде всего, конечно, нам потребуется код этого приложения. Откройте текстовый редактор (вполне подойдет Блокнот), наберите в нем код, представленный на рис. 1.12, и сохраните полученный текстовый файл как TestApp.cs.

Теперь наша задача — превратить этот исходный код в готовое приложение. При этом нам придется указать компилятору, какое именно приложение мы хо-

тим получить на выходе — консольное (с расширением EXE), графическое Windows (опять-таки с расширением EXE), в виде модуля DLL или какое-либо другое. Для этого мы должны будем при компиляции указать в виде параметра командной строки нужный нам флаг. Перечень флагов компиляции для csc.exe представлен в табл. 1.5.

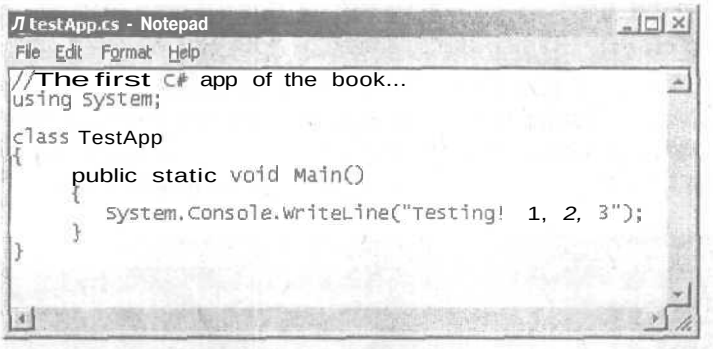


Рис. 1.12. Класс TestApp

Таблица 1.5. Флаги компиляции компилятора командной строки C#

Параметр командной строки	Значение
/doc	Комментарии из вашего исходника будут записаны в файл в формате XML (подробнее об этом — в главе 5)
/out	Указывается имя создаваемого двоичного файла (например, MyAssembly.dll, WordProcessingApp.exe и т. п.). Если этот параметр будет опущен, то имя создаваемого файла будет таким же, как у файла с исходным кодом (с расширением *.cs)
/target:exe	Тип создаваемого файла — консольное приложение. Этот тип принимается по умолчанию (то есть если параметр /target будет пропущен, будет создано именно консольное приложение)
/target: library	Будет создана сборка в виде библиотеки DLL. При этом эта сборка будет содержать манифест
/target:module	Двоичный модуль получится также в виде DLL, но уже без манифеста. Этот параметр используется только при создании многофайловых сборок
/target:winexe	Будет создано стандартное графическое («оконное») приложение Windows, В отличие от параметра /target:exe, при запуске полученного приложения командная строка открываться не будет

Поскольку по смыслу TestApp.cs — это консольное приложение, командная строка компилятора для него должна выглядеть следующим образом (обратите внимание, что флаг компиляции /target ставится до указания имени файла с исходным кодом, но не после):

```
csc /target:exe TestApp.cs
```

Параметр /target можно сократить до /t. Тогда командная строка компилятора будет выглядеть так:

```
csc /t:exe TestApp.cs
```

Как мы помним (из табл. 1.5), вообще-то, консольное приложение создается компилятором C# по умолчанию. Поэтому командная строка компилятора может выглядеть совсем просто:

```
csc TestApp.cs
```

На этом месте мы предлагаем вам перейти в тот каталог, в который вы поместили исходный файл TestApp.cs, и выполнить команду на компиляцию в любом из приведенных выше вариантов. В итоге должно получиться приложение C# с именем TestApp. Результат его выполнения представлен на рис. 1.13.

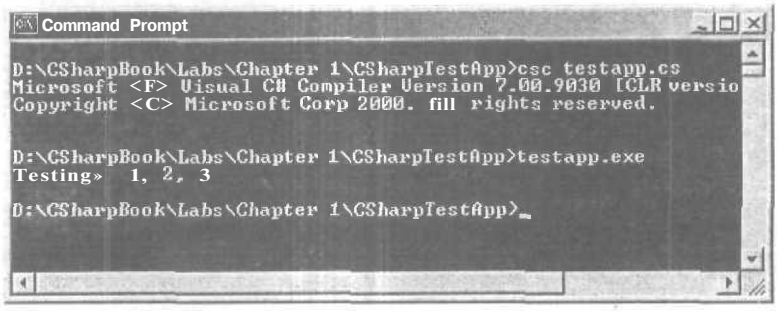


Рис. 1.13. Приложение TestApp можно сдавать заказчику

Ссылки на внешние сборки

Бывает так, что создаваемое приложение использует типы, которые находятся во внешних, совершенно отдельных сборках C#. В приложении TestApp проблем не возникло по той причине, что используемый нами класс System.Console был расположен в библиотеке базовых типов mscorlib.dll, к которой компилятор обращается автоматически. Однако иногда нужные нам классы расположены в других сборках. Ситуацию, которая при этом возникает, лучше продемонстрировать на примере.

Предположим, что наше приложение TestApp не только выводит строки на системную консоль, но и генерирует графическое окно сообщения (message box). Код TestApp при этом может выглядеть так, как показано на рис. 1.14.

Обратите внимание, что теперь в коде приложения мы обращаемся к классу в пространстве имен System.Windows.Forms. Чтобы компилятор смог обнаружить класс MessageBox, нам потребуется явно указать внешнюю сборку System.Windows.Forms.dll в командной строке. Для указания внешних сборок используется параметр /reference : имя_внешней_сборки (/reference можно сократить до /r). В нашем случае командная строка компилятора должна выглядеть так:

```
csc /r:System.Windows.Forms.dll TestApp.cs
```

При запуске нового варианта нашего приложения откроется диалоговое окно, такое, как на рис. 1.15.

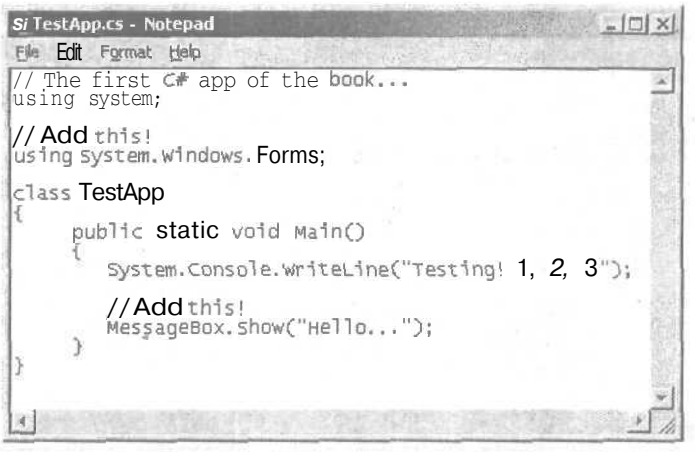


Рис. 1.14. Новый вариант TestApp.cs

Рис. 1.15. Ваше первое приложение Windows.Forms

Компиляция нескольких исходных файлов

При создании всех предыдущих вариантов нашего приложения использовался единственный исходный файл. Однако на практике гораздо чаще исходный код распределен по нескольким файлам. Давайте создадим такую ситуацию сами. Создадим дополнительный исходный файл HeLLoMsg.cs с классом HelloMessage, как показано на рис. 1.16, а наш файл TestApp.cs изменим в соответствии с рис. 1.17.

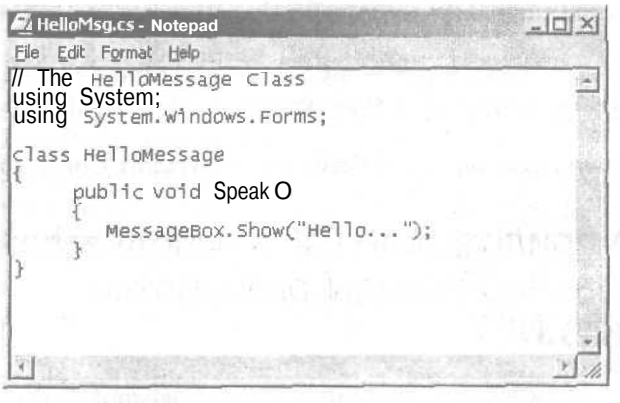


Рис. 1.16. Исходный файл с классом HelloMessage

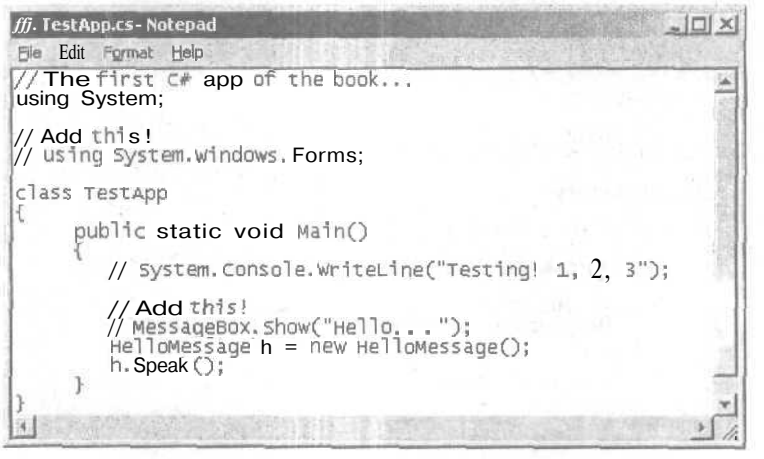


Рис. 1.17. Модифицированный файл TestApp.cs

При компиляции приложения **мы** можем явно указать все исходные файлы:

```
csc /r:System.Windows.Forms.dll TestApp.cs hellomsg.cs
```

Однако есть и еще один способ компилировать приложение из нескольких исходников. Компилятор csc.exe поддерживает символ подстановки (*). При использовании этого символа мы сообщаем компилятору, что ему необходимо включить в приложение все файлы с указанным нами расширением, которые находятся в текущем каталоге:

```
csc /r:System.Windows.Forms.dll /out:TestApp.exe *.cs
```

Поскольку компилятор уже не сможет воспользоваться именем приложения по умолчанию, при использовании подстановочного символа (*) нам придется обязательно использовать параметр /out.

А что делать, если в нашем приложении используются классы сразу из нескольких внешних сборок? Просто перечислить все эти сборки через точку с запятой, вот так:

```
csc /r:System.Windows.Forms.dll;System.Drawing.dll *.cs
```

В нашем приложении сборка System.Drawing.dll не нужна, мы добавили ее лишь для примера.

Как вы, наверное, догадываетесь, в CSC предусмотрено множество других параметров командной строки. Для знакомства с ними мы рекомендуем использовать MSDN.

Код приложения TestApp можно найти в подкаталоге Chapter 1.

Создание приложений C# с использованием интегрированной среды разработки Visual Studio.NET

Последнее, с чем мы должны познакомиться в этой вводной главе, — это основные возможности среды разработки Visual Studio.NET. Ключевое слово в предыдущей

фразе — «основные». Со многими другими возможностями среды Visual Studio вы познакомитесь в остальных главаxx

Первое, что необходимо сказать об интегрированной среде разработки (integrated development environment, IDE) Visual Studio.NET, — то, что эта среда теперь едина для всех языков программирования .NET от Microsoft. Таким образом, какой бы тип проекта вы ни создавали (ATL, MFC, C#, Visual Basic.NET, FoxPro, стандартный C++ и т. п.), вы все равно будете работать в одной и той же среде.

Начало работы со средой Visual Studio.NET

Прежде всего, конечно, надо запустить Visual Studio.NET и в меню File выбрать New, а затем Project Как мы можем убедиться, взглянув на рис. 1.18, типы проектов сгруппированы (по большей части) по используемым языкам программирования. Давайте для наглядности создадим в среде разработки Visual Studio.NET версию нашей программы, которую назовем VSNetTestApp.

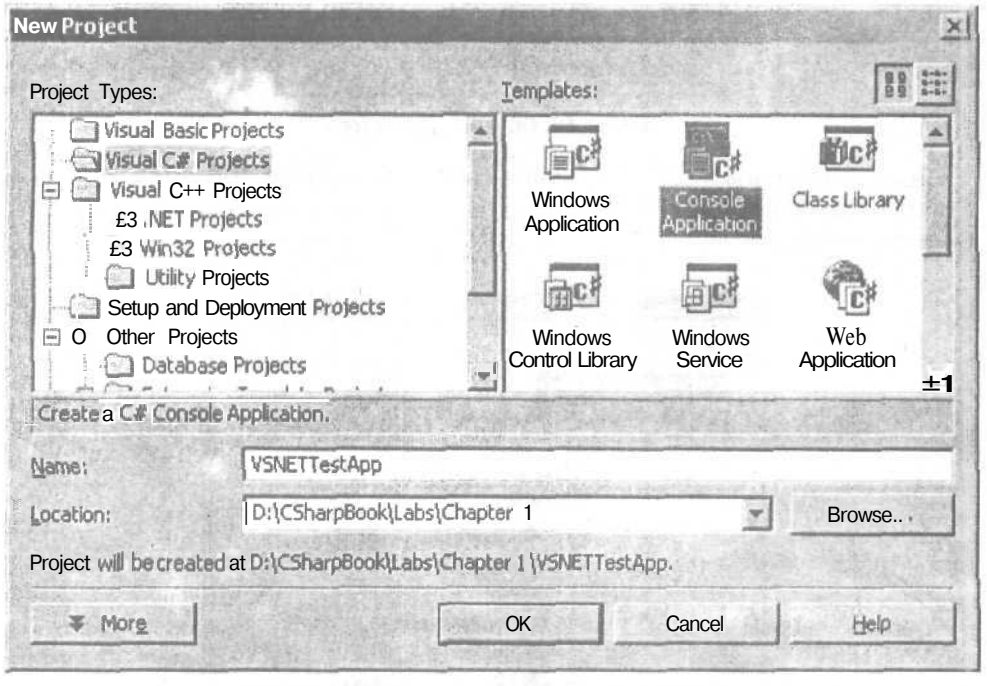


Рис. 1.18. Создание нового проекта VS.NET C#

Окно Solution Explorer

В интегрированной среде разработки Visual Studio.NET проекты логически организуются в *решения* (solutions). Каждое решение состоит из одного или нескольких проектов. В свою очередь, каждый проект может состоять из любого количества исходных файлов, ссылок на внешние сборки и прочих ресурсов, которые и образую г приложение. Вы сможете открыть любой из данных ресурсов с помощью окна Solution

Explorer - Проводника решений (рис. 1.19). Обратите внимание, что по умолчанию вашему первому классу в приложении присваивается имя Class1.cs.

Рис. 1.19. Окно Solution Explorer

Помимо обычной вкладки Solution Explorer, в этом окне также предусмотрена вкладка Class View — для «объектно-ориентированного» отображения иерархии вашего приложения (рис. 1.20).

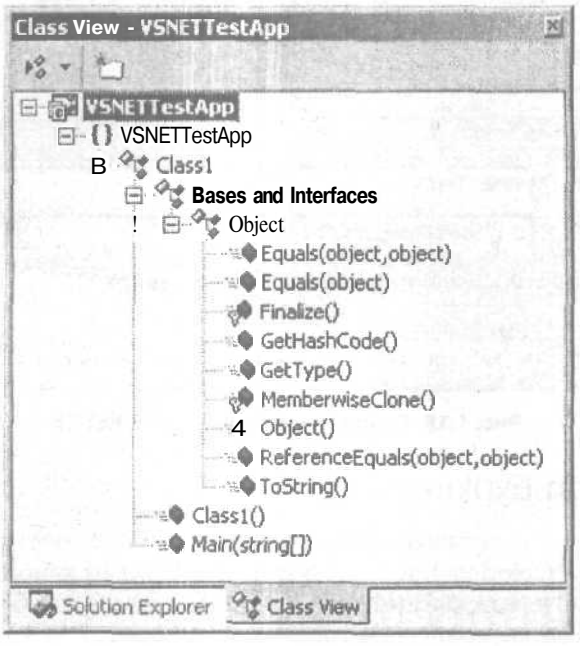


Рис. 1.20. Вкладка Class **View**

При щелчке правой кнопкой мыши на любом элементе в окне Solution Explorer в вашем распоряжении будет контекстное меню (рис. 1.21), с помощью которого БЫ сможете воспользоваться любым из множества CASE-средств, например, для добавления в ваш тип новых членов — методов, свойств, полей и т. д.

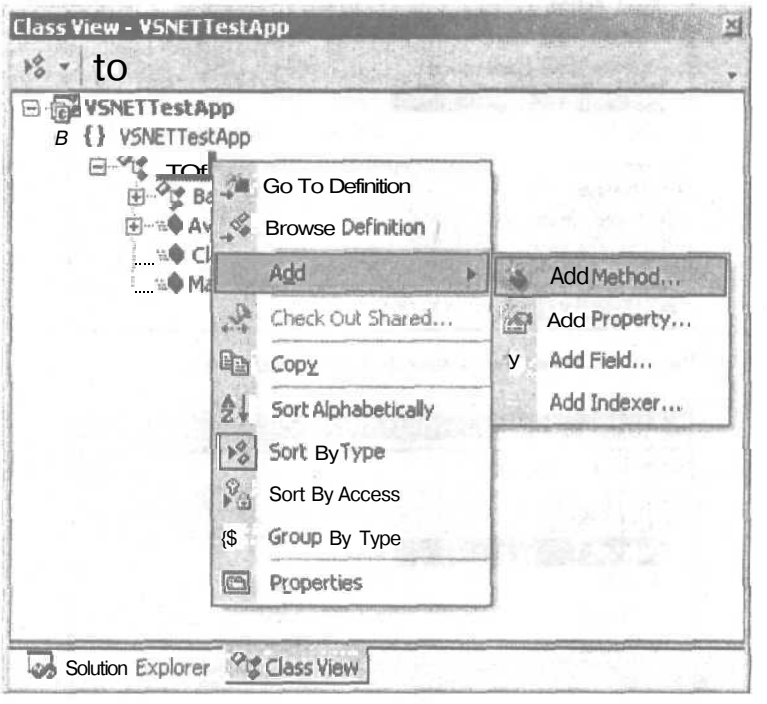


Рис. 1.21. Интегрированные мастера Visual Studio.NET

Я рекомендую вам опробовать все. В этой книге практически весь код мы будем писать вручную — для лучшего понимания языка, однако после создания дюжины-другой приложений почему бы и не воспользоваться интегрированным мастерами, если это позволяет сэкономить время.

Окно Properties

Другой важный элемент интегрированной среды разработки — это окно Properties (Свойства). В этом окне отображаются важнейшие характеристики выделенного в настоящий момент элемента. Этим элементом может быть и исходный файл, и элемент управления графического интерфейса пользователя, и проект в целом. Например, чтобы изменить имя исходного файла, выберите его в окне Solution Explorer и измените значение свойства FileName в окне Properties (рис. 1.22).

Изменение имени класса производится точно так же — просто выберите нужный класс на вкладке Class View и выполните необходимые действия в окне Properties (рис. 1.23). Обратите внимание, что изменения автоматически будут произведены во всем вашем коде.

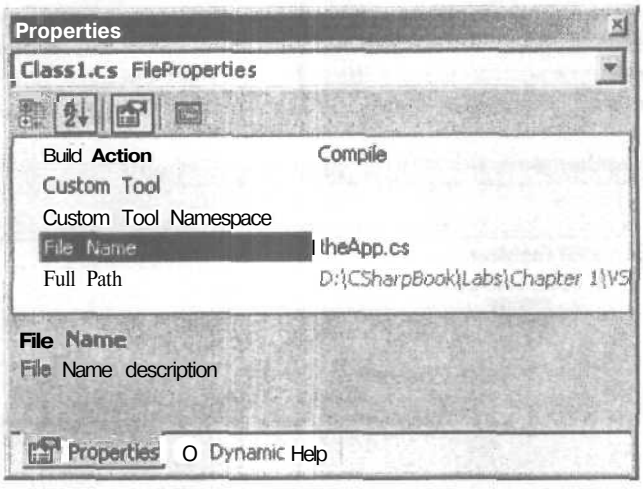


Рис. 1.22. Изменение имени файла с помощью окна Properties

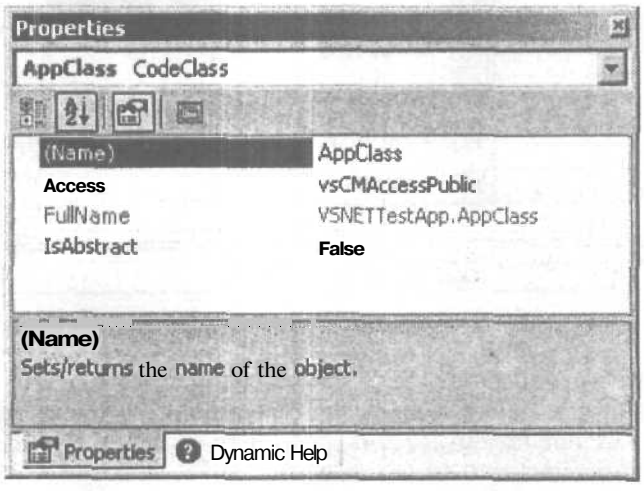


Рис. 1.23. Изменение имени класса с помощью окна Properties

Экономное размещение кода на экране

Еще одна полезная особенность среды разработки Visual Studio.NET — это возможность при отображении сжимать и разжимать участки программного кода, как при работе с обычной иерархией из дерева и узлов (рис. 1.24). Нажатие на значок «+» приводит к открытию участка программного кода для выбранного вами элемента, нажатие на значок «-» позволяет скрыть данный отрезок кода, освободив место на рабочем столе. При наведении курсора мыши на многоточие, означающее скрытый для удобства код, данный код будет показан во всплывающем окне.

Конечно же, в среде разработки Visual Studio.NET предусмотрена полная поддержка технологии IntelliSense, «подсказывающей» вам в то время, когда вы набираете код, и предлагающей закончить за вас начатую строку (рис. 1.25).

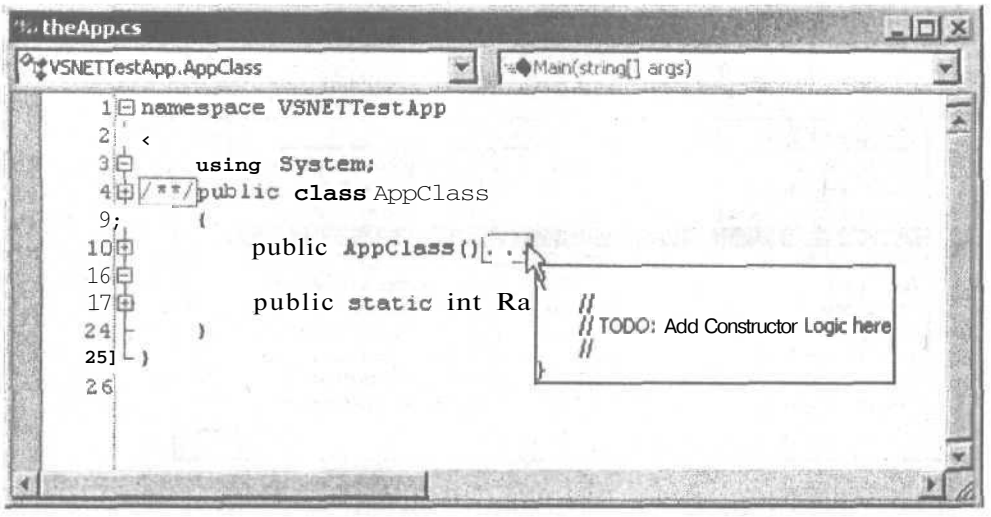


Рис. 1.24. Часть кода можно скрыть

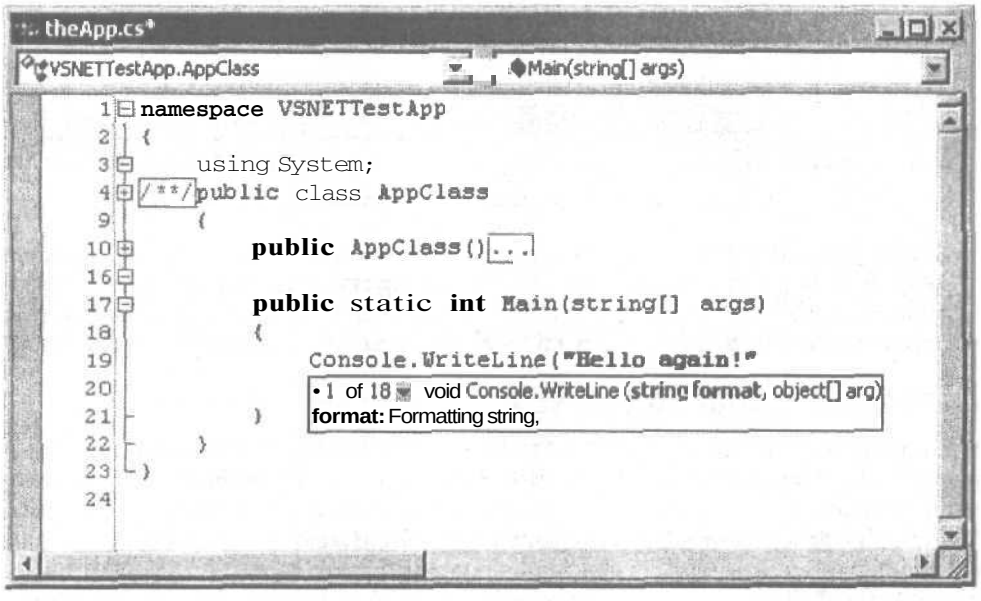


Рис. 1.25. Работает технология IntelliSense

Ссылки на внешние сборки

Если вам потребовалось добавить ссылку на внешнюю сборку (в нашем примере это была библиотека System.Windows.Forms.dlL) в ваш текущий проект, вы можете сделать это двумя способами: либо воспользоваться меню Project ▸ Add Reference, либо сделать то же самое через контекстное меню для узла Assembly (Сборка) в окне Solution Explorer, Б любом случае должно открыться диалоговое окно, представленное на рис. 1.26.

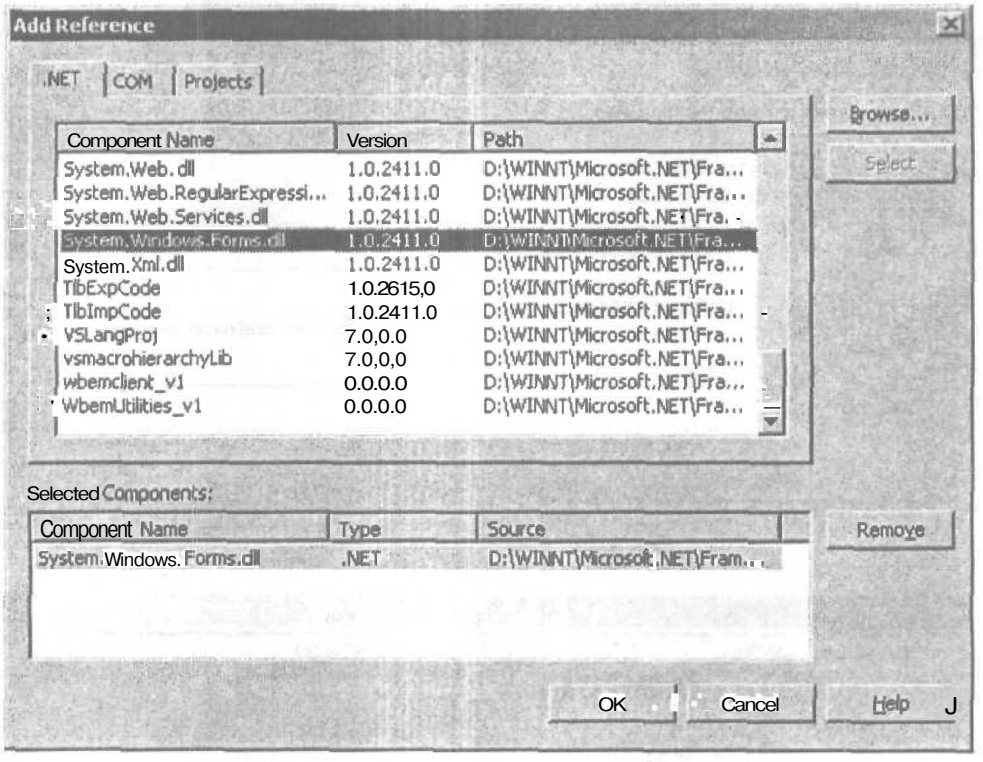


Рис. 1.26. Добавление ссылок на внешние сборки

Из рисунка видно, что в проект можно добавлять ссылки не только на внешние сборки .NET, но и на двоичные файлы COM, и на другие проекты (подробнее о захватывающих аспектах взаимодействия .NET/COM будет рассказано в главе 12). Для нашего приложения в этом окне нужно добавить ссылку на System.Windows.Forms.dll.

Отладка в Visual Studio.NET

Как и в предыдущих версиях Visual Studio, в Visual Studio.NET, конечно же, предусмотрен интегрированный отладчик. Добавить контрольную точку можно, щелкнув мышью на самом левом сером столбце в окне кода (там, где на рис. 1.27 темный кружок).

При работе в режиме отладки выполнение приложения будет прерываться на каждом брейкпойнте. С помощью панели инструментов Debug (Отладка) вы можете производить пошаговое выполнение (и также возвращаться назад). В интегрированном отладчике предусмотрено множество специальных окон (Call Stack, Autos, Locals, Breakpoints, Modules, Exceptions и т. п.). Чтобы скрывать ненужные вам окна, и наоборот, открывать нужные, используйте меню Debug ▸ Windows.

Код проекта VSNETTestApp можно найти в подкаталоге Chapter 1.

Работа с окном Server Explorer

Еще одно очень важное средство Visual Studio.NET называется Server Explorer. Окно Server Explorer можно открыть с помощью меню View (рис. 1.28).

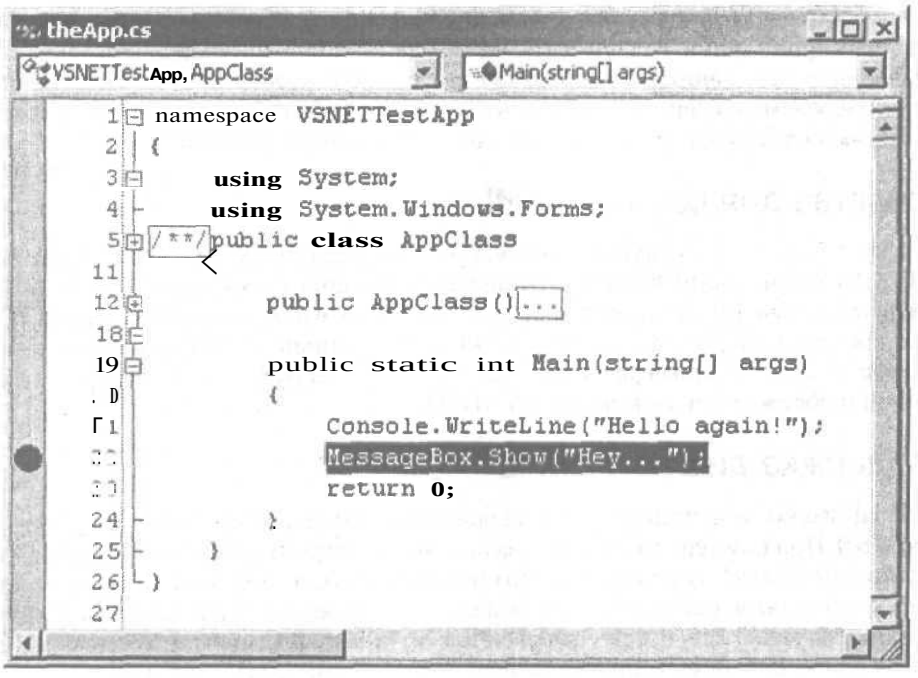


Рис. 1.27. Установка брейкпойнтов

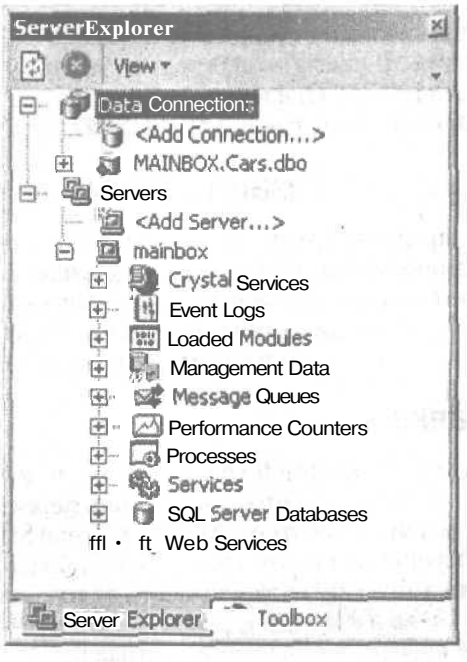


Рис. 1.28. Окно Server Explorer

Server Explorer можно рассматривать как командный центр при создании распределенных приложений. Используя Server Explorer, вы сможете подключать локальные и удаленные базы данных и выполнять в них различные операции, организовать работу с очередями сообщений, работать с журналом событий, получать информацию о работающих службах и производить множество других действий.

Средства для работы с XML

В Visual Studio.NET предусмотрены встроенные средства для работы с XML (как и HTML). Многие из этих средств были унаследованы от прежнего Visual InterDev. После подключения (или создания) файла XML к вашему приложению вы сможете производить редактирование его кода при помощи множества графических средств. В качестве примера на рис. 1.29 показан файл XML, который будет создан нами при обсуждении возможностей ADO.NET.

Поддержка диаграмм UML

В Visual Studio.NET получил свое дальнейшее развитие Visual Modeler из Visual Studio 6.0. При помощи этого средства вы сможете строить диаграммы UML (Unified Modeling Language) типов вашего приложения. Для того чтобы воспользоваться возможностями диаграмм UML, необходимо добавить в приложение файл *.mdx (с помощью меню File ▸ Miscellaneous Files ▸ File...). В вашем распоряжении появятся новые инструментальные панели с элементами UML (рис. 1.30).

Утилита Object Browser

Помимо тех трех отдельных утилит, которые были рассмотрены ранее в этой главе, для просмотра информации о типах вы можете воспользоваться встроенным просмотрщиком объектов Visual Studio.NET. Окно Object Browser открывается через меню View ▸ Other Windows ▸ Object Browser. Пример окна Object Browser представлен на рис. 1.31.

Средства для работы с базами данных

В Visual Studio.NET предусмотрены встроенные средства для организации соединений с базами данных. Как уже говорилось выше, после того как вы настроите подключение к базе данных в окне Server Explorer, вы сможете производить с этой базой данных и ее объектами любые операции. На рис. 1.32 представлена база данных Cars, с которой мы будем работать в главе, посвященной ADO.NET.

Встроенная справка

Последний аспект работы с Visual Studio.NET, о котором нельзя не упомянуть, — это встроенная справка. Вместо, того чтобы постоянно переключаться с помощью клавиш Alt+Tab между средой разработки и MSDN, в Visual Studio.NET можно воспользоваться предусмотренным для этих целей окном Dynamic Help. Содержимое этого окна меняется (динамически!) в зависимости от того, какой именно элемент (окно, меню, ключевое слово в коде и т. п.) выделен в настоящий момент. Например, если вы поместите курсор на объявление метода Main(), то окно Dynamic Help примет вид, представленный на рис. 1.33.

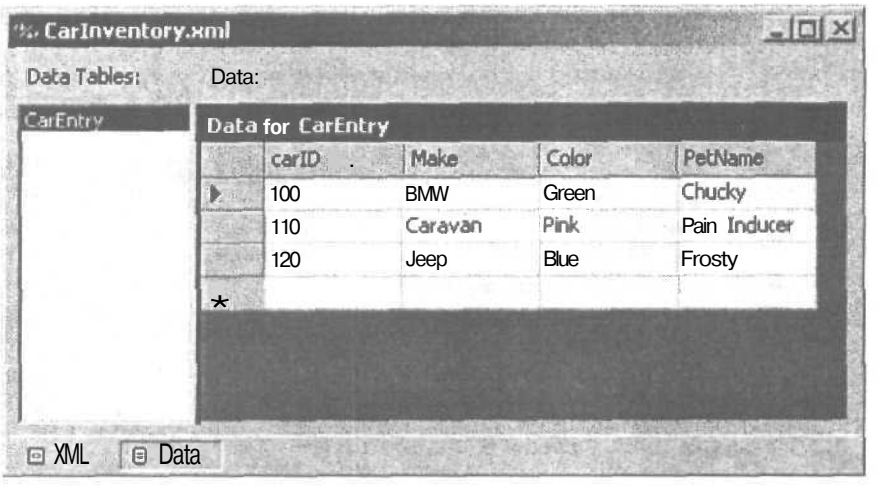


Рис. 1.29. Встроенный редактор XML

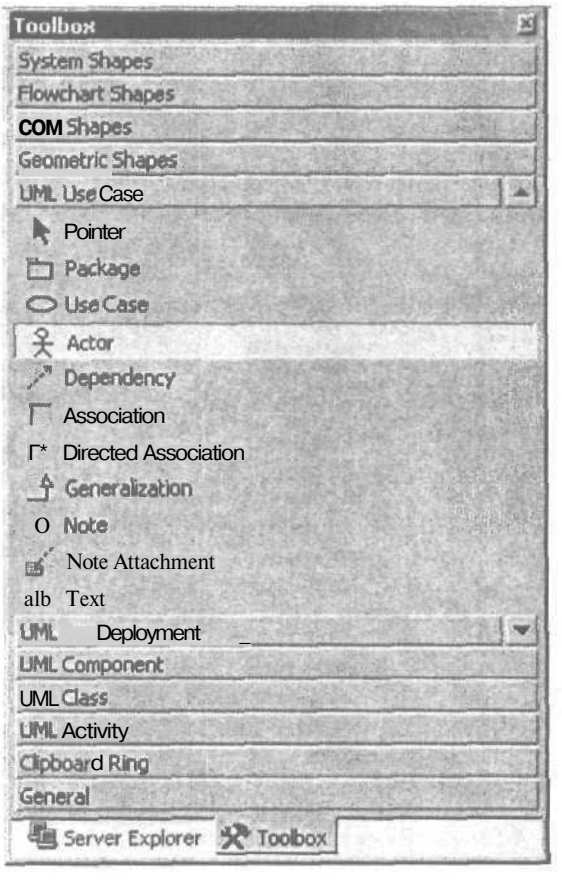


Рис. 1.30. Интегрированные средства UML

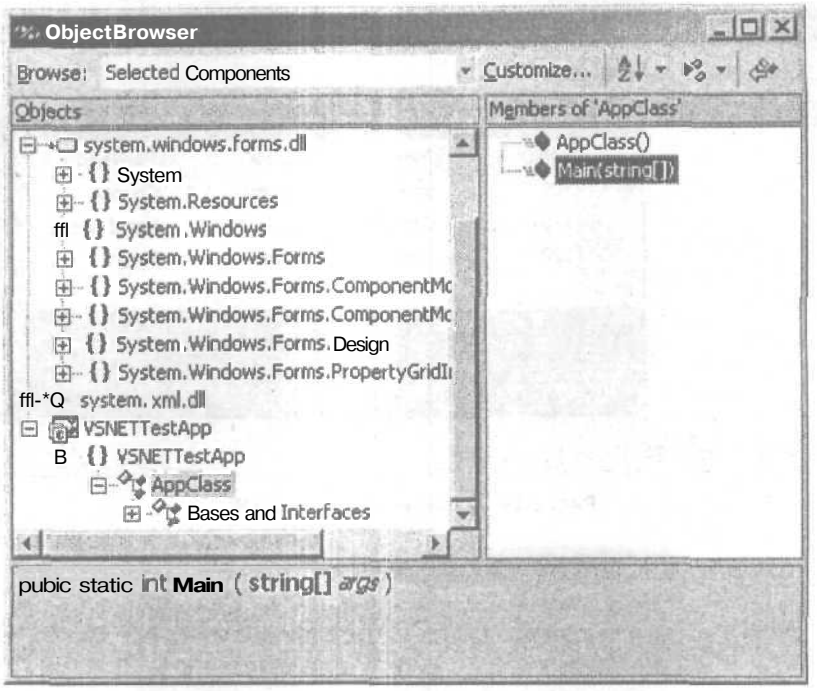


Рис. 1.31. Встроенная утилита Object Browser

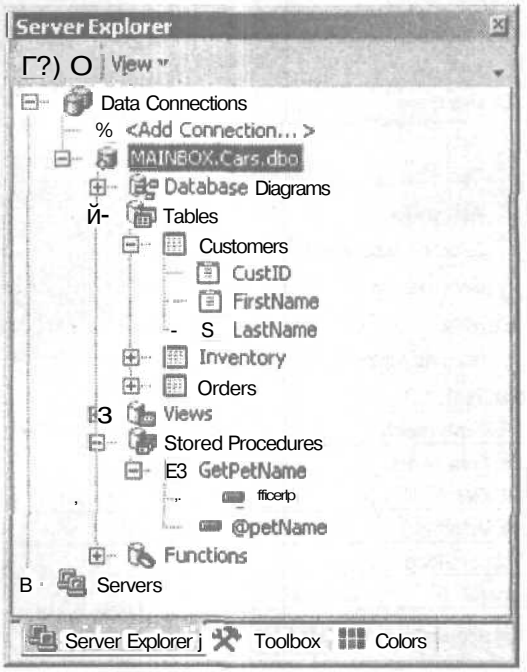


Рис. 1.32. Интегрированные средства для работы с базами данных

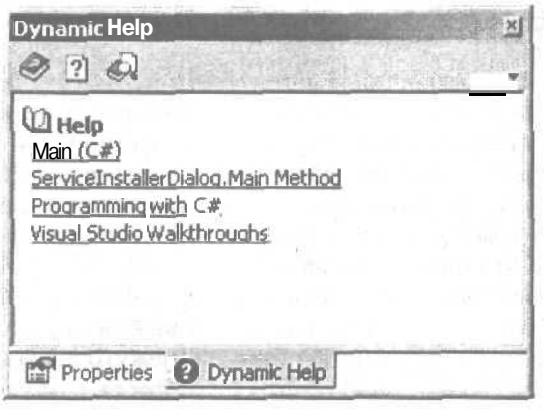


Рис. 1.33. Окно Dynamic Help

Конечно же, по щелчку на гиперссылке в окне Dynamic Help откроется связанная с ней информация (рис. 1.34).

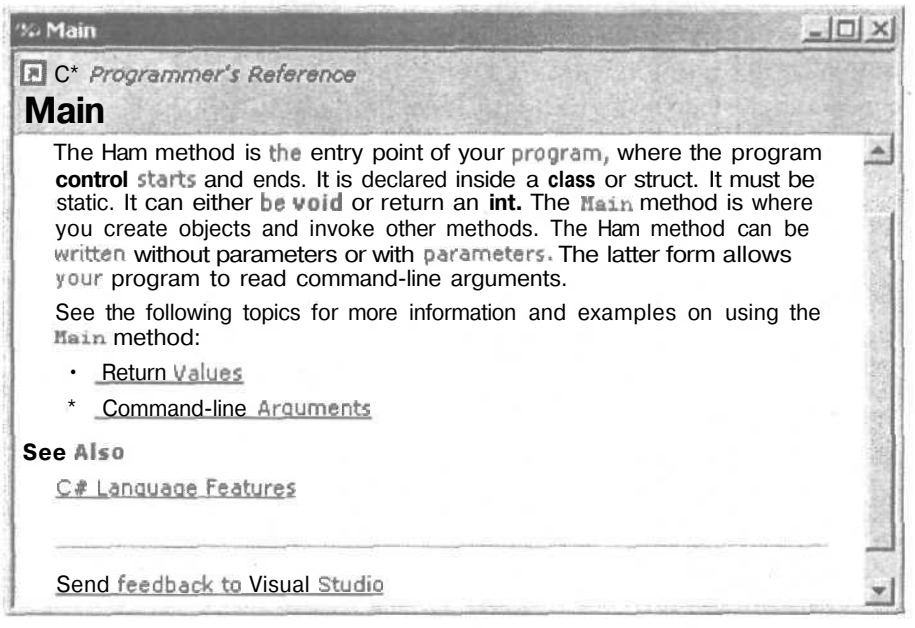


Рис. 1.34. Одна из страниц встроенной справки Visual Studio.NET

Как вы могли убедиться, в вашем распоряжении множество средств — и новых, и не очень.

Теперь, когда вы приобрели обширные знания относительно философии .NET и умеете компилировать приложения C# по крайней мере двумя способами, самое время приступить к формальному изучению C# и платформы .NET с точки зрения программирования.

Подведение итогов

Главная задача этой главы состояла в том, чтобы дать вам общие представления, необходимые для изучения материала всех последующих глав этой книги. В самом ее начале были рассмотрены те проблемы и ограничения, с которыми столкнулись существующие в настоящее время традиционные подходы к программированию, и то, какие пути решения этих проблем предлагает платформа .NET.

Два важнейших компонента платформы .NET — это среда выполнения .NET (Common Language Runtime, CLR), воплощенная в файле mscoree.dll, и библиотека базовых классов (mscorlib.dll и остальные файлы). Любые двоичные файлы .NET, называемые «сборками» (assemblies), работают только в среде выполнения .NET – CLR. Сборки содержат в себе инструкции промежуточного кода (Intermediate Language, IL) и метаданные самой сборки (манифест) и типов. Промежуточный код превращается в платформенно-зависимый код в момент выполнения приложения. Эту задачу выполняет JIT(just-in-time compiler), компилятор времени выполнения. Помимо этого в Visual Studio.NET в единую систему сведены все допустимые типы данных. Для этого используются концепции CTS (Common Type System) и CLS (Common Language Specification). В конце главы вы познакомились с утилитами, входящими в состав .NET SDK, с компилятором командной строки C# — csc.exe и основными возможностями интегрированной среды разработки Visual Studio.NET.

Основы языка C# 2

В этой главе мы познакомимся с основными аспектами C# как языка программирования, встроенными типами данных, конструкциями условных переходов и циклов, механизмами упаковки и распаковки, а также со средствами создания простых классов. Кроме того, мы научимся работать средствами C# с символьными значениями, массивами, перечислениями и структурами.

Чтобы наглядно показать приемы работы с этими средствами языка, мы постоянно будем обращаться к библиотекам базовых классов C# и создадим несколько приложений с использованием различных системных пространств имен. Помимо работы с уже готовыми пространствами имен мы с вами освоим способы организации ваших типов в собственные, создаваемые вами пространства имен (а также выясним, в каких ситуациях это может оказаться полезным).

Анатомия класса C#

C# похож на язык Java в том отношении, что он требует, чтобы вся программная логика была заключена в определения типов (вспомним, что под типом подразумеваются классы, интерфейсы, структуры и аналогичные компоненты языка). В отличие от C (и C++) глобальные функции и глобальные переменные в чистом виде в C# использовать нельзя. Простейший класс C# может быть определен следующим образом:

```
// Обычно для файлов классов C# используется расширение *.cs
using System:

class HelloClass
{
        // Как ни удивительно, функция Main() может быть объявлена как private,
        // если, конечно, вам это нужно...
        public static int Main (string[] args)
        {
                Console.WriteLine ("Hello, World");
```

```
            return 0;
        }
}
```

Мы создали класс HelloClass, который поддерживает единственный метод — Main(). В любом приложении C# должен быть определен класс с методом Main(). Этот метод будет использован как точка входа вашего приложения — то есть работа приложения начнется с выполнения этого метода. В принципе, возможно создать несколько классов с методами Main(), однако в этом случае придется явно указать, какой из этих методов будет использован в качестве точки входа. Если вы пропустите такое указание, результатом станет сообщение компилятора об ошибке.

Обратите внимание на то, что первая буква в названии метода Main() — заглавная. Такое написание обязательно.

В нашем случае метод Main() был определен как public и как static. Подробнее об этих ключевых словах будет рассказано дальше в этой главе. Пока достаточно будет сказать, что ключевое слово public в определении метода означает, что этот метод будет доступен извне, а ключевое слово static говорит о том, что этот метод позиционируется на уровне класса, а не отдельного объекта и будет доступен даже тогда, когда еще не создано ни одного экземпляра объекта данного класса.

Кроме того, наш метод Main() принимает единственный параметр, который должен быть набором символов (string[] args). Хотя этот параметр в нашей программе нигде не встречается, его можно использовать для приема параметров командной строки при запуске приложения.

Вся программная логика HelloClass заключена в самом методе Main(). Этот метод обращается к классу Console, определенному внутри пространства имен System. Один из членов метода Console — статический метод WriteLine(), который можно использовать для вывода строки на системную консоль;

```
// Выводим строку на системную консоль
Console.WriteLine("Hello, World");
```

Поскольку метод Main() был определен нами как int, то он должен вернуть целочисленное значение (в нашем случае 0 — стандартный код успешного завершения программы), Последнее, что мы должны сказать, завершая рассмотрение нашей программы, — что в C# поддерживаются комментарии в стиле C и C++.

Как еще можно объявить метод Main()

В нашем примере метод Main() был объявлен как возвращающий целочисленное значение (int) и принимающий единственный параметр в виде массива символов. Однако метод Main() можно объявлять и по-другому. Вот несколько вариантов допустимых сигнатур метода Main() (помните, что все они должны находиться только внутри определения класса):

```
// Без возвращаемого значения, принимаемый параметр - массив символов
public static void Main (string[] args)
{
        // Обрабатывем параметры командой строки
        // Создаем объекты
}

// Без возвращаемого значения, принимаемых параметров нет
```

```
public static void Main()
{
        // Создаем объекты
}

// Возвращается целочисленное значение, принимаемых параметров нет
public static int MainO
I
        // Создаем объекты
        // Возвращаем целочисленное значение операционной системе
}
```

Какой же вариант объявления Main() следует выбирать? Все зависит от ответов на два вопроса. Первый вопрос: должна ли ваша программа принимать какие-либо параметры командной строки? Если да, то придется объявлять Main() с принимаемым параметром в виде массива символов. И второй вопрос: должно ли значение возвращаться операционной системе при завершении работы программы (то есть по завершении метода Main())? Если ответ положительный, то метод MainC) должен быть объявлен как int.

Обработка параметров командной строки

Предположим, что вы хотите изменить класс HelloClass таким образом, чтобы он не игнорировал параметры командной строки, а обрабатывал их. После внесения необходимых изменений HelloClass может выглядеть так (подробнее о конструкции {0} мы расскажем чуть позже):

```
using System;

class HelloClass
{
        public static int Main (string[] args)
        {
                // Выводим параметры на консоль!
                for(int x=0; x < args.Length; x++
                {
                        Console.WriteLine("Arg: {0}", args[x]);
                }

                Console.WriteLine("Hello, World!");
                return 0;
        }
}
```

В нашей программе производится проверка, были ли вообще переданы какие-либо значения в качестве параметров командной строки. Такая проверка производится с помощью свойства Length объекта System.Array (как мы потом убедимся, в C# все массивы — это лишь псевдонимы для объекта System.Array, и они обладают такими же наборами членов). Если в массиве есть по крайней мере один член (параметр командной строки), запускается цикл, в ходе которого на консоль выводятся все параметры командной строки с первого до последнего.

Результат выполнения нашей программы может выглядеть так, как показано на рис. 2.1.

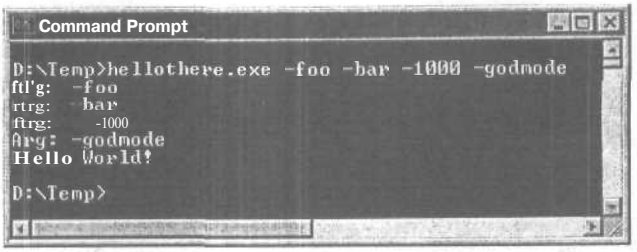


Рис. 2.1. Программа обрабатывает переданные ей параметры командной строки

Цикл для последовательной обработки всех параметров командной строки можно организовать и другим способом, использовав конструкцию foreach (подробнее о ней будет рассказано ниже в этой главе):

```
public static int Main (string[] args)
{
        foreach (string s in args)
                Console.WriteLine("Arg: {0}", s);
        ...
}
```

Естественно, использовать или не использовать параметры командной строки — зависит только от программиста, создающего приложение.

Создание объектов: конструкторы

Во всех объектно-ориентированных языках четко различаются понятия «класс» и «объект». Класс — это определяемый пользователем тип данных (user-defined type, UDT). Класс можно представить себе как чертеж, по которому организуются его члены. В отличие от класса объектом называется конкретный экземпляр определенного класса, с помощью которого обычно и производятся определенные действия в программе.

Единственный способ создания нового объекта в C# — использовать ключевое слово new. Давайте рассмотрим, как создаются объекты, на следующем примере:

```
// Объекты HelloClass будут созданы при вызове метода Main()
Using System;
class HelloClass
{
        public static int Main(string[] args)
        {
                // Новый объект можно создать в одной строке
                HelloClass c1 = new HelloClass();

                // ... или в двух:
                HelloClass c2;
                c2 - new HelloClassO;

                return 0:
        }
}
```

Ключевое слово new означает, что среде выполнения следует выделить необходимое количество оперативной памяти под экземпляр создаваемого объекта. Вы-

деление памяти производится из «кучи», находящейся в распоряжении среды выполнения .NET (managed heap — управляемой кучи). В нашем примере в оперативной памяти созданы два объекта c1 и c2, каждый из которых является экземпляром типа HelloClass. В C# переменные класса — это ссылки на объект в памяти, но не сами объекты. Поэтому c1 и c2 — это ссылки на два отдельных объекта, находящиеся в оперативной памяти (точнее, в области управляемой кучи). Подробнее об этом будет рассказано в главе 3.

В нашем примере объекты создаются при помощи *конструктора по умолчанию*. Компилятор C# автоматически снабжает конструктором по умолчанию любой класс. Если есть необходимость, вы можете переопределить этот конструктор, заставив его выполнять нужные вам действия. Как и в C++, конструктор класса по умолчанию — это конструктор, который не принимает никаких параметров. Однако в C# имеется существенное отличие от C++ в отношении конструктора по умолчанию. В C++ при создании объекта при помощи конструктора по умолчанию переменные остаются неинициализированными (то есть в них могут быть любые случайные значения). В C# конструктор по умолчанию (как и все другие конструкторы), если не указано иное, присваивают всем данным состояния (например, переменным-членам) значения по умолчанию — в большинстве случаев 0. Кажется, что отличие небольшое, но в некоторых ситуациях это свойство конструкторов C# убережет вас от ошибок.

В большинстве случаев класс помимо конструктора по умолчанию снабжается и другими конструкторами — принимающими параметры. Использование такого конструктора — самый простой способ инициализировать состояние объекта в момент его создания, установив нужные вам значения. В качестве примера мы еще раз обратимся к классу HelloClass. При этом мы придадим этому классу несколько переменных-членов, значения которых можно настраивать, конструктор с параметрами, а также переопределим конструктор по умолчанию — конструктор без параметров.

```
// HelloClass с конструкторами
using System;

class HelloClass
{
        // Конструктор по умолчанию присвоит переменным-членам значения по умолчанию
        public HelloClass()
        {
                Console.WriteLine("Default ctor called!");
        }

        // Наш собственный конструктор присвоит переменным-членам специальные значения
        public HelloClass(int x, int y)
        {
                Console.WriteLine("Custom ctor called!");
                intX = x;
                intY = y;
        }

        // Объявляем переменные-члены класса
        public int intX, intY;

        // Точка входа для программы
        public static int Main(string[] args)
```

```
    {
        // Применяем конструктор по умолчанию
        HelloClass c1 - new HelloClass();
        Console.WriteLine("c1.intX = {0}\nc1.intY - {1}\n", c1.intX, cI.intY);

        // Применяем конструктор с параметрами
        HelloClass c2 - new HelloClass(100, 200);
        Console.WriteLine("c2.intX = {0}\nc2.intY - {1}\n", c2.intX, c2.intY);

        return 0;
    }
}
```

Запустив эту программу, можно убедиться, что конструктор по умолчанию действительно присвоил переменным-членам значения по умолчанию (нулевые), а конструктор с параметрами присвоил переменным-членам значения, соответствующие этим параметрам (рис. 2.2).

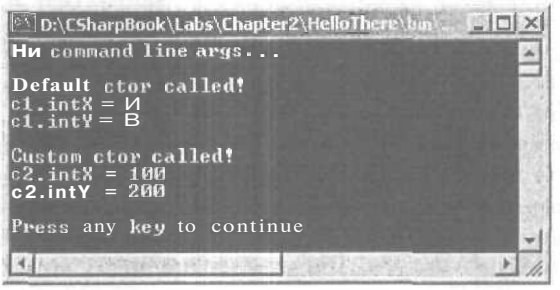


Рис. 2.2. Результат применения конструкторов

Будет ли происходить утечка памяти?

Обратите внимание, что в программной логике метода Main() не предусмотрено никаких средств для удаления объектов cl и c2:

```
// Метод с утечкой памяти?
public static int Main(string[] args)
{
    HelloClass cI - new HelloClassO:
    Console.WriteLine("c1.intX - {0}\nc1.intY = {1}\n", cI.intx. cI.intY):
    HelloClass c2 = new HelloClass(100, 200):
    Console.WriteLine("c2.intX - {0}\nc2.intY - {1}\n", c2.intX. c2.intY);

    // Ау! Не забыли ли мы удалить кое-какие объекты?
    return 0:
}
```

Ничего ужасного не произошло. Напротив, в .NET так и надо поступать. В C#, как и в Visual Basic, и в Java, программистам не надо думать о явном удалении объектов и освобождении памяти — сборщик мусора .NET освободит память автоматически. Поэтому в C# нет даже зарезервированного слова del ete. Подробнее о работе сборщика мусора .NET будет рассказано в следующей главе.

Композиция приложения C#

Наш класс HelloClass выполняет две основные функции: во-первых, он обеспечивает точку входа для приложения, а во-вторых, он содержит в себе две переменные-члены класса и два перегруженных конструктора. Несмотря на то что все работает просто замечательно, может показаться странным, что метод Main() создает экземпляр того самого класса, в котором он определен:

```
class HelloClass
{
        public HelloClass(){ Console.WriteLine("Default ctor called!"); }

        public HelloClass (int x. int y)
        {
                Console.WriteLine ("Custom ctor called!");
                intX - x; intY - y:
        }

        public int intX. intY;

        public static int Main(string[] args)
        {
                // Создаем объекты HelloClass...
                HelloClass cl - new HelloClass();
                ...
        }
}
```

Этот подход уже несколько раз использовался в наших примерах и будет использован еще неоднократно. Однако при создании реальных приложений правильно использовать другой подход, а именно разбить класс HelloClass на два отдельных класса: HelloClass и HelloApp. В терминах объектно-ориентированного программирования такой принцип получил название «разделения интересов» (separation of concerns). Следуя ему, мы можем преобразовать наше приложение следующим образом (обратите внимание, что в HelloClass появился новый метод):

```
class HelloClass
{
        public HelloClass(){ Console.WriteLine("Default ctor called!"); }
        public HelloClass (1nt x, int y)
        {
                Console.WriteLine ("Custom ctor called!");
                intX - x; intY - y;
        }
        public int intX. intY;

        // Новая функция-член
        public void SayHi() {Console.WriteLine("Hi there!");}
}

class HelloApp
{
        public static int Main{string[] args)
        {
                // Создаем объекты HelloClass и выводим приветствие
                HelloClass cl = new HelloClass();
```

```
            c1.SayHi();
            ...
        }
}
```

При создании приложений принято использовать схему, в которой один класс (класс приложения) представляет объект приложения, помимо всего прочего, обеспечивая точку входа для этого приложения, а логика выполнения различных операций заключена в других классах. При этом обычно для каждого класса используется отдельный файл *,cs (что может сильно облегчить повторное использование вашего кода). В приложениях-примерах в нашей книге мы будем использовать оба подхода.

Код приложения HelloThere можно найти в подкаталоге Chapter 2.

Инициализация членов

Иногда для класса может быть предусмотрено множество конструкторов. При этом может возникнуть такая ситуация, когда вам придется при создании класса инициализировать переменные, присваивая им одни и те же значения вне зависимости от того, какой именно конструктор вызывается. Если эти значения отличны от значений по умолчанию, то записывать одинаковые строки инициализации переменных в каждый вариант конструктора — значит, делать много лишней работы.

C# избавляет вас от нее. В нем вы можете инициализировать переменные прямо в момент их объявления:

```
class Text
{
        private int MyInt - 90;
        private string MyString = "My initial value";
        private HotRod viper = new HotRod (200, "Chucky", Color.Red);
        ...
}
```

Другие объектно-ориентированные языки, такие как C++, не позволяют инициализировать члены классов таким способом. Конечно, программисты нашли выходы из этой ситуации. Например, можно создать свою маленькую вспомогательную функцию, присваивающую переменным нужные значения, и затем вызывать эту функцию из каждого конструктора. Еще один возможный подход — передавать вызовы от каждого из вариантов конструктора «главному» конструктору (мы будем подробнее говорить об этом в главе 3 при обсуждении зарезервированного слова thi s). Все эти подходы, конечно же, можно использовать и в C#, но возможность присваивать значения переменным прямо при их объявлении, по крайней мере, не менее удобна.

Ввод и вывод с использованием класса Console

В большинстве созданных нами приложений использовался класс System.Console — один из многих классов, определенных внутри пространства имен System. Как уже можно было заметить, этот класс предназначен для работы с вводом и выводом (в том числе сообщений об ошибках) с использованием системной консоли (которую иногда также называют командной строкой). Вы вряд ли ошибетесь, если предположите, что этот класс чаще всего используется в консольных приложениях .NET.

Главные методы класса Console — это методы ReadLine() и WriteLine() (оба этих метода определены как статические). WriteLine(), как мы неоднократно могли убедить-

ся, выводит символьную строку (дополняя ее в конце символами перехода на новую строку и возврата каретки) на системную консоль. Метод Write() делает то же самое, но уже без дополнения символами перехода на новую строку. Метод ReadLine() позволяет считать информацию с системной консоли до ближайшего символа переходи на новую строку, метод Read() считывает с системной консоли единственный символ.

Мы проиллюстрируем применение различных методов класса Console на примере. В нем пользователю будет предложено ввести некоторую информацию, которая сразу же будет выведена вновь на системную консоль.

```
// Применяем класс Console для ввода и вывода данных
using System;
class BasicIO
{
        public static void Main(string[] args)
        {
                // Выводим приглашение ввести имя
                Console.Write("Enter your name: ");

                string s;
                s - Console.Readline();
                Console.WriteLine("Hello, {0}", s);
                Console.Write("Enter your age: ");

                s = Console.ReadLine();
                Console.WriteLine("You are {0} years old\n", s):
        }
}
```

Результаты работы программы представлены на рис. 2.3.

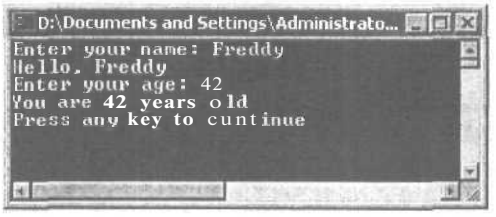


Рис. 2.3. Ввод и вывод данных при помощи класса System.Console

Средства форматирования строк в C#

Начиная с самых первых примеров, вы постоянно встречали в коде одни и те же метки {0}, {1} и т. д. В C# предусмотрены новые средства форматирования символьных строк, отчасти напоминающие старую добрую функцию C printf(), однако без загадочных флагов %d, %s, %c и остальных. Применение этих меток проще проиллюстрировать еще одним примером:

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

```
        int theInt = 90;
        float theFloat = 9.99;
        BasicIO myIO = new BasicIO();

        // Комбинируем символьную строку:
        Console.WriteLine("Int is: {0}\nFloat is: {1}\nYou are: {2}",
        theInt, theFloat, myIO.ToString());
    }
}
```

Результат выполнения программы представлен на рис. 2.4.

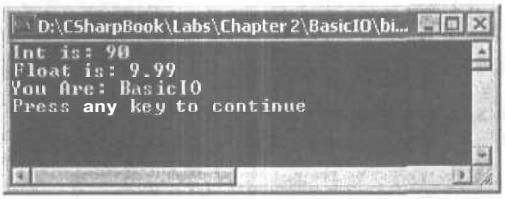


Рис. 2.4. Результат выполнения класса BasicIO

Первый параметр, передаваемый `WriteLine()`, представляет собой строку форматирования с подстановочными выражениями {0}, {1}, {2} и т. п. Остальные параметры `WriteLine()` — это как раз те значения, которые последовательно подставляются в места, обозначенные подстановочными выражениями. Создатели библиотеки базовых типов позаботились о перегрузке метода `WriteLine()` таким образом, что в качестве второго параметра этого метода можно передавать массив объектов. При этом подстановочные выражения будут указывать на элементы этого массива:

```
object[] stuff = { "Hello", 20.9, 1, "There", "83", 99.99933 };
Console.WriteLine("The Stuff: {0}, {1}, {2}, {3}, {4}, {5}", stuff);
```

В каждом подстановочном выражении при желании можно использовать параметры форматирования, представленные в табл. 2.1 (эти параметры можно записывать как строчными, так и прописными буквами).

Таблица 2.1. Параметры форматирования C#

Параметр	Значение
C или c	Используется для вывода значений в денежном (currency) формате. По умолчанию перед выводимым значением подставляется символ доллара ($), хотя можно отменить подстановку этого символа при помощи объекта NumberFormatInfo
D или d	Используется для вывода десятичных значений. После этого символа можно указать количество выводимых символов после запятой
E или e	Для вывода значений в экспоненциальном формате
F или f	Вывод значений с фиксированной точностью
G или g	Общий (general) формат. Применяется для вывода значений с фиксированной точностью или в экспоненциальном формате
N или n	Стандартное числовое форматирование с использованием разделителей (запятых) между разрядами
X или x	Вывод значений в шестнадцатеричном формате. Если вы использовали прописную X, то буквенные символы в шестнадцатеричных символах также будут прописными

Символы форматирования следуют в подстановочных выражениях сразу же за номером подставляемого параметра через двоеточие: {0:C}, {1:d}, {2:X} и т. д. Изменим содержание метода Main() еще раз:

```
// Применяем параметры форматирования
public static void Main(string[] args)
{
...
        Console.WriteLine("C format: {0:C}", 99989,987);
        Console.WriteLine("D9 format: {0:D9}", 99999);
        Console.WriteLine("E format: {0:E}", 99999.76543);
        Console.WriteLine("F format: {0:F3}", 99999,9999);
        Console.WriteLine("N format: {0:N}", 99999);
        Console.WriteLine("X format: {0:X}", 99999);
        Console.WriteLine("x format: {0:x}", 99999);
}
```

Результаты выполнения этой программы представлены на рис. 2.5.

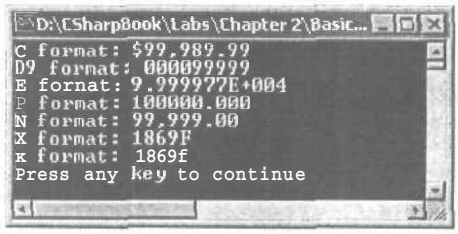


Рис. 2.5. Примеры использования параметров форматирования

Конечно, параметры форматирования C# можно применять не только с методом System.Console.WriteLine(). Те же самые параметры можно использовать для форматирования любых символьных строк, например при использовании статического метода String.Format(). Такое средство может оказаться полезным при создании в памяти строки символов с числовыми значениями и последующим отображением этой строки любым способом:

```
// Используем статический метод String.Format() для создания новой символьной строки
string formStr;
formStr = String.Format("Don't you wish you had {0:C} in your account?", 99989,987)
Console.WriteLine(FormStr);
```

Код приложения BasicIO можно найти в подкаталоге Chapter 2.

Структурные и ссылочные типы

Как в любом языке программирования, в C# существует множество встроенных типов данных. С помощью этих типов можно представлять целочисленные числа и числа с плавающей запятой, строки символов и логические значения. Если вы пришли из мира C++, возможно, вас обрадует то, что в мире .NET разрядность всех встроенных типов фиксирована и постоянна. Поэтому, если вы используете тип данных 1nt, вы можете быть уверенными, что этот тип будет абсолютно одинаков в любом .NET-совместимом языке программирования.

Все типы в C# разделяются на две основные разновидности: структурные типы (value-based) и ссылочные типы (reference-based). К структурными типам относятся все числовые типы данных (int, float и пр.), а также перечисления и структуры. Память для структурных типов выделяется из стека. При присвоении одного структурного типа другому присваивается не сам тип (как область в памяти), а его побитовая копия. Давайте рассмотрим это на примере такой структуры C# (подробнее о структурах будет рассказано дальше в этой главе):

```
// Структуры относятся к структурным типам
struct FOO
{
        public int x,y;
}
```

Теперь мы используем эту структуру в следующей программе:

```
class ValRefClass
{
        // Экспериментируем со структурными типами
        public static int Main(string[] args)
        {
                // При создании структуры с применением конструктора по умолчанию
                // использование ключевого слова "new" является необязательным
                FOO f1 = new FOO();
                f1.x = 100;
                f1.y = 100;

                // Присваиваем новому типу FOO (f2) существующий тип FOO (f1)
                FOO f2 = f1;

                // Выводим содержимое f1
                Console.WriteLine("F1.x = {0}", f1.x);
                Console.WriteLine("F1.y = {0}", f1.y);

                // Выводим содержимое f2
                Console.WriteLine("F2.x = {0}", f2.x);
                Console.WriteLine("F2.y = {0}", f2.y);

                // А теперь вносим изменения в f2.x. Они не отразятся на f1.x
                Console.WriteLine ("Changing f2.x");
                f2.x=900;

                // Снова выводим содержимое
                Console.WriteLine("F2.x = {0}", f2.x);
                Console.WriteLine("F1.x = {0}", f1.x);
                return 0;
        }
}
```

Результаты работы программы представлены на рис. 2.6.

Мы создали объект типа FOO с именем f1, который затем был отнесен к другому одному типу FOO и получил имя f2. Поскольку FOO — это структурный тип, в результате в специальной области оперативной памяти — стеке были созданы две копии структуры FOO, каждая из которых дальше живет своей собственной отдельной жизнью. Поэтому при изменении значения f2.x значение f1.x осталось прежним.

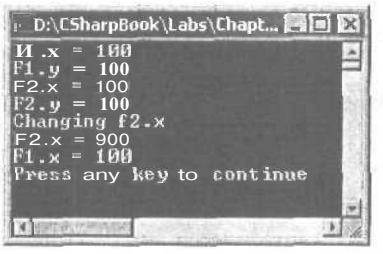


Рис. 2.6. При присвоении одного структурного типа другому происходит присвоение не самого типа, а его полной (побитовой) копии

Ссылочные типы (классы и интерфейсы) ведут себя совершенно по-другому. Память для них выделяется не в стеке, а в области управляемой кучи. При копировании ссылочного типа создается еще одна ссылка, которая указывает на ту же область оперативной памяти. Для того чтобы наглядно это продемонстрировать, изменим определение FOO типа, достаточно заменить одно-единственное слово. Мы превратим структуру FOO в ссылочный тип — класс FOO:

```
// Классы - это всегда ссылочный тип
class FOO
{
        public int x, y;
}
```

Теперь наша программа работает совсем по-другому, как это показано на рис. 2.7.

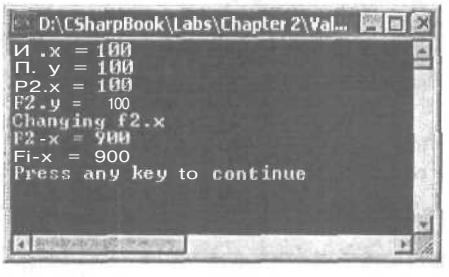


Рис 2.7. При присвоении одного ссылочного типа другому оба типа начинают указывать на одну и ту же область оперативной памяти

Поскольку и f1, и f2 теперь — не более чем ссылки на один и тот же диапазон памяти в области управляемой кучи, изменение x через ссылку f2 приводит к тому, что значение x, получаемое через ссылку f1, также изменяется.

Структурные и ссылочные типы: исследуем дальше

Прочие отличия структурных и ссылочных типов для наглядности мы сведем в одну таблицу (табл. 2.2). Многие из интригующих вопросов будут потом рассмотрены более подробно.

Таблица 2.2. Сравнение структурных и ссылочных типов

«Интригующий вопрос»	Структурные типы	Ссылочные типы
Где размещаются эти типы?	В области стека	В области управляемой кучи
Как будет представлена переменная, которой в качестве значения присвоен этот тип?	В виде локальной копии типа	В виде указателя на область оперативной памяти, относящейся к объекту этого типа
Что может являться для них базовым типом?	Эти типы могут производиться только напрямую от типа System.ValueType	Могут производиться от любого другого типа (за исключением System.ValueType), если этот тип не является закрытым
Может ли этот тип выступать в качестве базового?	Нет, Структурные типы всегда закрыты и дополнение их другими свойствами не предусмотрено. Поэтому они не могут выступать при наследовании в качестве базового для других типов	Да, если этот ссылочный тип не определен внутренне как закрытый
Как производится передача параметров?	Как значений (то есть вызывающей функции передаются только локальные копии значений переменных)	Как ссылок (то есть вызывающей функции передается ссылка на адрес данного объекта в оперативной памяти)
Существует ли возможность переопределить Object.Finalize()?	Нет. Структурные типы никогда не размещаются в куче и поэтому к ним не применяется функция завершения	Да, но не напрямую (подробнее об этом — в главе 3)
Можно ли определить конструкторы для этого типа?	Да, но конструктор по умолчанию зарезервирован (то есть все ваши конструкторы должны принимать параметры)	Конечно!
Когда происходит «смерть» переменной данного типа?	Когда происходит выход за область видимости	Во время процесса сборки мусора (garbage collection) в управляемой куче

Несмотря на все различия, и структурные, и ссылочные типы могут реализовывать стандартные (то есть встроенные) и пользовательские (создаваемые вами) интерфейсы и могут поддерживать любое количество полей, методов, свойств и событий. Чтобы еще раз наглядно продемонстрировать различия между структурными и ссылочными типами, можно рассмотреть следующий код:

```
// Структурный тип
struct PERSON
{
        public string Name;
        public int Age;
};
// Ссылочный тип
class Person
```

```
f
        public string Name:
        public int Age;
};
class ValRefClass
{
        public static void Main()
        {
                // Создаем ссылку на объект в управляемой куче
                Person fred = new Person();

                // Создаем значение в стеке
                PERSON tagy = new PERSONO;

                // Создаем еще один объект в стеке - точную копию первого объекта
                PERSON jane = tagy;

                // А здесь происходит создание всего лишь еще одной ссылки
                // на уже существующий объект в оперативной памяти
                Person fredRef = fred;
        }
]
```

Код приложения ValAndRef можно найти в подкаталоге Chapter 2.

Точка отсчета для любых типов: System.Object

В C# все типы данных (как структурные, так и ссылочные) производятся от единого общего предка: класса System.Object. Класс System.Object определяет общее полиморфическое поведение для всех типов данных во вселенной .NET (можете считать, что System.Object — эта некоторая разновидность типа Variant в .NET). Во всех предыдущих примерах у нас не было необходимости явно указывать класс System.Object в качестве базового — это подразумевается само собой. Однако нам ничто не мешает сделать это, явнс указав, что наш класс производится от System.Object:

```
// Еще один вариант: написать "class HelloClass ; object"
class HelloClass : System.Object
{.--}
```

Как и в любом другом классе C#, в классе System.Object существует свой набор членов. Обратите внимание, что некоторые члены определены как виртуальные и должны быть замещены в определении производного класса:

```
// Самый верхний класс в иерархии классов .NET: System.Object
namespace System
{
        public class Object
        {
                public Object();
                public virtual Boolean Equals (Object obj);
                public virtual Int32 GetHashCode();
                public Type GetType();
                public virtual String ToString();
```

```
        protected virtual void Finalize();
        protected Object MemberwiseClone();
    }
...
}
```

Предназначение каждого из методов описано в табл. 2.3.

Таблица 2.3. Главные методы объекта System.Object

Метод	Назначение
Equals()	По умолчанию этот метод возвращает «истинно» только тогда, когда сравниваемые сущности указывают на одну и ту же область в оперативной памяти. Поэтому этот метод в его исходной реализации предназначен только для сравнения объектов ссылочных типов, но не структурных. Для нормальной работы с объектами структурных типов этот метод необходимо заместить. Однако помните, что если вы замещаете этот метод, вам потребуется также заместить метод GetHashCode()
GetHashCode()	Возвращает целочисленное значение, идентифицирующее конкретный экземпляр объекта данного типа
GetType()	Метод возвращает объект Type(), полностью описывающий тот объект, из которого метод был вызван. Это метод идентификации времени выполнения (Runtime Type Identification, RTTI), который предусмотрен во всех объектах (подробнее об этом — в главе 7)
ToString()	Возвращает символьное представление объекта в формате <имя_пространства_имен>.<имя_класса> (такой формат носит также название «полностью определенного имени» — fully qualified name). Если тип определен вне какого-либо пространства имен, возвращается только имя класса. Этот метод может быть замещен для представления информации о внутреннем состоянии объекта (в формате имя — значение)
Finalize()	Пока мы будем считать, что основное назначение этого метода — освободить все ресурсы, занятые объектом данного класса, перед удалением этого объекта. Подробнее о службах сборки мусора в CLR будет рассказано в главе 3
MemberwiseClone()	Этот метод предназначен для создания еще одной ссылки на область, занимаемую объектом данного типа в оперативной памяти. Этот метод не может быть замещен. Если вам потребовалось реализовать поддержку создания полной копии объекта в оперативной памяти, вы должны реализовать в вашем классе поддержку интерфейса ICloneable (об этом — в главе 4)

Мы продемонстрируем использование методов, которые унаследованы от System.Object, при помощи следующего определения класса:

```
// Создаем объекты и знакомимся с методами, унаследованными от System.Object
using System;
class ObjTest
{
    public static int Main(string[] args)
    {
        // Создаем экземпляр ObjTest
        ObjTest c1 = new ObjTest();

        // Выводим информацию на консоль
        Console.WriteLine("ToString: {0}", c1.ToString());
        Console.WriteLine("Hash Code: {0}", c1.GetHashCode());
```

```
            Console.WriteLine("Type: {0}", c1.GetType().ToString());

            // Создаем еще одну ссылку на c1
            ObjTest c2 - cl;
            object o - c2;

            // Действительно ли все три экземпляра указывают на одну
            // и ту же область в оперативной памяти?
            if(o.Equals(c1) && c2.Equals(o))
                  Console.WriteLine("Same instance!");

            return 0;
      }
}
```

Результаты работы программы представлены на рис. 2.8,

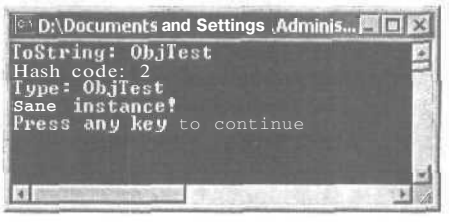


Рис. 2.8. Применение некоторых методов, унаследованных от System.Object

Обратите внимание, что реализация по умолчанию метода ToStri ng возвращает только имя того типа, из которого данный метод был вызван (ObjTest). Очень часто этот метод в производных классах замещается таким образом, чтобы возвращать информацию о внутреннем состоянии объекта (мы тоже этим вскоре займемся). А пока рассмотрим следующий отрезок кода:

```
// Сравниваем ссылки на объекты
public static int Main(string[] args)
{
      // Создаем экземпляр объекта ObjTest
      ObjTest cl = new ObjTest();
      ...
      // Создаем дополнительные ссылки на cl
      ObjTest c2 - cl:
      object o = cl:

      // Действительно ли все три экземпляра указывают на одну и ту же область
      // в оперативной памяти?
      if(o.Equals(c1) && c2.Equals(o))
            Console.WriteLine("Same instance!");

      return 0:
}
```

Как мы помним, исходный вариант Equals() использует для сравнения ссылочную, а не структурную семантику, В начале мы создали новый объект типа ObjTest с именем cl. В результате для этого объекта в области управляемой кучи была выделена оперативная память. Объект c2 также относится к типу ObjTest. Однако при его создании мы не создавали нового экземпляра объекта. Вместо этого была создана еще одна

ссылка на ту же область оперативной памяти, которую занимает объект c1. Точно таким же способом была создана и третья ссылка на ту же область памяти — объект o. Поскольку и cl, и c2, и o — это ссылки на одну и ту же область оперативной памяти, операция сравнения с использованием метода Equals() вернула положительный результат.

Замещение методов System.Object

Методы, которые типы данных наследуют от System.Object, во многих ситуациях исключительно полезны. Но обычно при создании своих собственных типов данных некоторые методы System.Object приходится замещать. Рассмотрим такое замещение на примере. Для целей нашего примера мы будем использовать уже знакомый нам класс Person, в который мы добавим переменные для хранения данных об имени человека, его номере социального страхования и возрасте:

```
// Помните! Все классы в конечном итоге производятся от класса System.Object
class Person
{
        public Person(string fname, string lname, string ssn, byte a)
        {
                firstName - fname:
                lastname = lname:
                SSN - ssn;
                age = a;
        }

        public Person(){} // Всем переменным-членам будут присвоены значения по умолчанию

        // Данные о человеке
        public string firstName:
        public string lastName;
        public string SSN:
        public byte age;
}
```

В качестве замещаемого метода мы выберем Object.ToString(). Мы хотим, чтобы этот метод возвращал не имя типа, а информацию о внутреннем состоянии объекта. Более подробно замещение методов будет рассмотрено в следующей главе. Сейчас нам достаточно отметить, что мы меняем поведение метода ToString() специально для нашего класса Person:

```
// Ссылка на это пространство имен необходима для получения доступа к типу StringBuilder
using System.Text;

// В нашем классе Person метод ToString будет реализован следующим образом:
class Person
{
...
        // Замещаем метод, унаследованный от System.Object:
        public override srting ToString()
        {
                StringBuilder sb - new StringBuilder()

                sb.Append("[FirstName= " + this.firstName);
                sb.Append(" LastName= " + this.lastName);
                sb.Append(" SSN= " + this.SSN);
                sb.Append(" Age- " + this.age + "]");
```

```
        return sb.ToString();
    }
```

Г

Каким будет конкретный формат данных, возвращаемых новым методом ToString(), — это во многом вопрос личных вкусовых предпочтений. В этом примере я предпочел заключить возвращаемые данные в квадратные скобки ([...]). Кроме того, в этом примере мы впервые использовали класс System.Text.StringBuilder (нам еще предстоит более подробное знакомство с этим классом). Однако выбор этого класса — также вопрос личных предпочтений. Вместо него вполне можно было бы использовать что-нибудь другое.

Следующий метод, замещением которого мы займемся, — метод Equals(). Как мы помним, этот метод возвращает истинно только в тех ситуациях, когда мы сравниваем ссылки на один и тот же объект в оперативной памяти. Во многих ситуациях имеет смысл заместить этот метод таким образом, чтобы он был полезен не только при работе со ссылочными типами, но и со структурными. Да и в отношении ссылочных типов иногда может потребоваться, чтобы этот метод вел себя не так, как его исходный вариант. Пусть наш метод Equals() возвращает истинно тогда, когда у сравниваемых объектов типа Person одинаковые внутренние состояния (то есть значения переменных firstName, lastName, SSN и age совпадают):

```
// Наш класс Person реализует метод Equals() таким образом:
class Person
{
...
    public override bool Equals (object o)
    {
        // Совпадают ли у объекта, принимаемого в качестве параметра,
        // значения переменных с иоими?
        Person temp - (Person)o;

        if (temp.firstName - = this.firstName &&
           <temp.lastName = =this.lastName &&
            temp.SSN = = this.SSN &&
            temp.age = = this.age)
                return true;
        else
                return false;
    }
...
!
```

Мы сравниваем значения переменных объекта, принимаемого в качестве параметра, со значениями объекта, из которого метод Equals() был вызван (через использование ключевого слова this). Если значения переменных firstName, lastNane, SSN и age одинаковы, то оба объекта обладают одинаковым внутренним состоянием и метод Equals() возвращает истинно.

Следует обратить внимание еще на один момент. Если мы замещаем метод Equals(), то мы должны также заместить метод GetHashCode(). В случае, если это сделано не будет, компилятор выдаст предупреждение. Метод GetHashCode() возвращает числовое значение, идентифицирующее объект в оперативной памяти. Чаще всего это значение используется в коллекциях, работающих с хэшами объектов.

Существует масса способов сгенерировать уникальный хэш для объекта (среди них встречаются весьма затейливые). Для наших целей вполне подойдет значение, возвращаемое тем же самым методом GetHashCode(), но для переменной SSN (ведь тип данных, который для нее использован, также является производным от System.Object, и поэтому этот метод применим):

```
// Возвращаем хэш-код, основанный на номере социального страхования (переменной SSN)
public override int GetHashCodeO
{
        return SSN.GetHashCode();
}
```

После этого наш класс Person готов к использованию. Его применение может выглядеть так:

```
// Создаем несколько объектов Person и экспериментируем с замещенными методами
public 1nt Main(string[] args)
{
        // Создаем несколько объектов и производим их сравнение на идентичность.
        // Мы специально присваиваем одинаковые значения переменным-членам для целей
        // тестирования метода Equals()
        Person p1 = new Person("Fred", "Jones", "222-22-2222", 98);
        Person p2 - new Person("Fred", "Jones", "222-22-2222", 98);

        // Используем замеченный для сравнения внутреннего состояния метод Equals()
        1f(pl.Equals(p2) && p1.GetHashCode() = = p2.GetHashCode())
                Console.WriteLine ("P1 and P2 have same state\n");
        else
                Console.WriteLine ("P1 and P2 are DIFFERENT\n");

        // Теперь меняем внутреннее состояние p2
        p2.age = 2;

        // Производим сравнение заново:
        if(p1.Equals(p2) && p1.GetHashCode() - - p2.GetHashCode())
                Console.WriteLine ("P1 and P2 have same state\n");
        else
                Console.WriteLine ("P1 and P2 are DIFFERENT\n");

        // Теперь используем замещенный метод ToString()
        Console.WriteLine(p1.ToString());
        Console.WriteLine(p2);        // Вообще-то, метод WriteLine() вызывает метод
                                      // ToStringO автоматически
        return 0;
}
```

Результат выполнения этой программы представлен на рис. 2.9.
Код приложения ObjectMethods можно найти в подкаталоге Chapter 2.

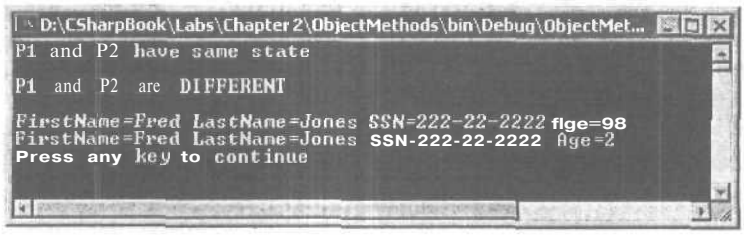


Рис. 2.9. Результат применения замещенных методов Equals() и ToStringO

Статические члены System.Object

Помимо тех членов, которые мы с вами рассмотрели, в System.Object имеются два статических члена, которые также являются исключительно полезными при сравнении объектов, особенно когда нужно отдельно производить сравнение внутреннего состояния объектов и сравнение адресов ссылок — указывают ли ссылки на одну и ту же область в оперативной памяти или нет. Рассмотрим следующий код:

```
// Статические члены System.Object
Person p3 = new Person("Sally", "Jones", "333", 4);
Person p4 = new Person("Sally", "Jones", "333", 4);

// Одинаково ли внутреннее состояние p3 и p4? Да!
Console.WriteLine("P3 and P4 have same state: {0}", object.Equals (p3, p4));

// Представляют ли они один и тот же обьект в оперативной памяти? Нет!
Console.WriteLine("P3 and P4 are pointing to same object: {0}",
                  object.ReferenceEquals(p3, p4));
```

Таким образом, используя эти варианты методов Equals() и ReferenceEquals(), вы можете просто передавать два объекта любого типа классу System.Object (не забывайте, что эти методы — статические) — все остальные операции по сравнению будут произведены автоматически.

Системные типы данных и псевдонимы *C#*

Как уже говорилось, любой встроенный тип данных C# — это всего лишь псевдоним для существующего типа, определенного в пространстве имен System. В табл. 2.4 перечислены все системные типы данных, соответствующие им псевдонимы C#, а также информация о совместимости данных типов с CLS.

Таблица 2.4. Системные типы данных и псевдонимы C#

Псевдоним C#	Соответствует ли CLS	Системный тип	Диапазон хранимой информации	Назначение
sbyte	Нет	SByte	От -128 до 127	Знаковое 8-битовое число
byte	Да	Byte	От 0 до 255	Беззнаковое 8-битовое число
short	Да	Int16	От 32 768 до 32 767	Знаковое 16-битовое число
ushort	Нет	UInt16	От 0 до 65 535	Беззнаковое 16-битовое числе
int	Да	Int32	От -2 147 483 648 ДО 2 147 483 647	Знаковое 32-битовое число
uint	Нет	UInt32	От 0 до 4 294 967 295	Беззнаковое 32-битовое число
long	Да	Int64	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	Знаковое 64-битовое число
ulong	Нет	UInt64	От О до 18 446 744 073 709 551 615	Беззнаковое 64-битовое число

продолжение

Таблица 2.4 (*продолжение*)

Псевдоним C#	Соответствует ли CLS	Системный тип	Диапазон хранимой информации	Назначение
char	Да	Char	От U+0000 до U+ffff	Один 16-битовый символ Unicode
float	Да	Single	От $1,5 \times 10^{-45}$ до $3,4 \times 10^{38}$	32-битовое значение с плавающей запятой
double	Да	Double	От $5,0 \times 10^{-324}$ до $1,7 \times 10^{308}$	64-битовое значение с плавающей запятой
bool	Да	Boolean	true или false	Представляет «истину» или «ложь»
decimal	Да	Decimal	От 10^{0} до 10^{28}	96-битовое знаковое значение
string	Да	String	Ограничено только системной памятью	Представляет набор символов Unicode
object	Да	Object	Практически все что угодно. Все типы происходят от System.Object, поэтому объектом является все	Класс, базовый для всех типов во вселенной .NET

Отношения между основными системными типами (а также теми, с которыми нам только предстоит познакомиться) представлены на рис. 2.10.

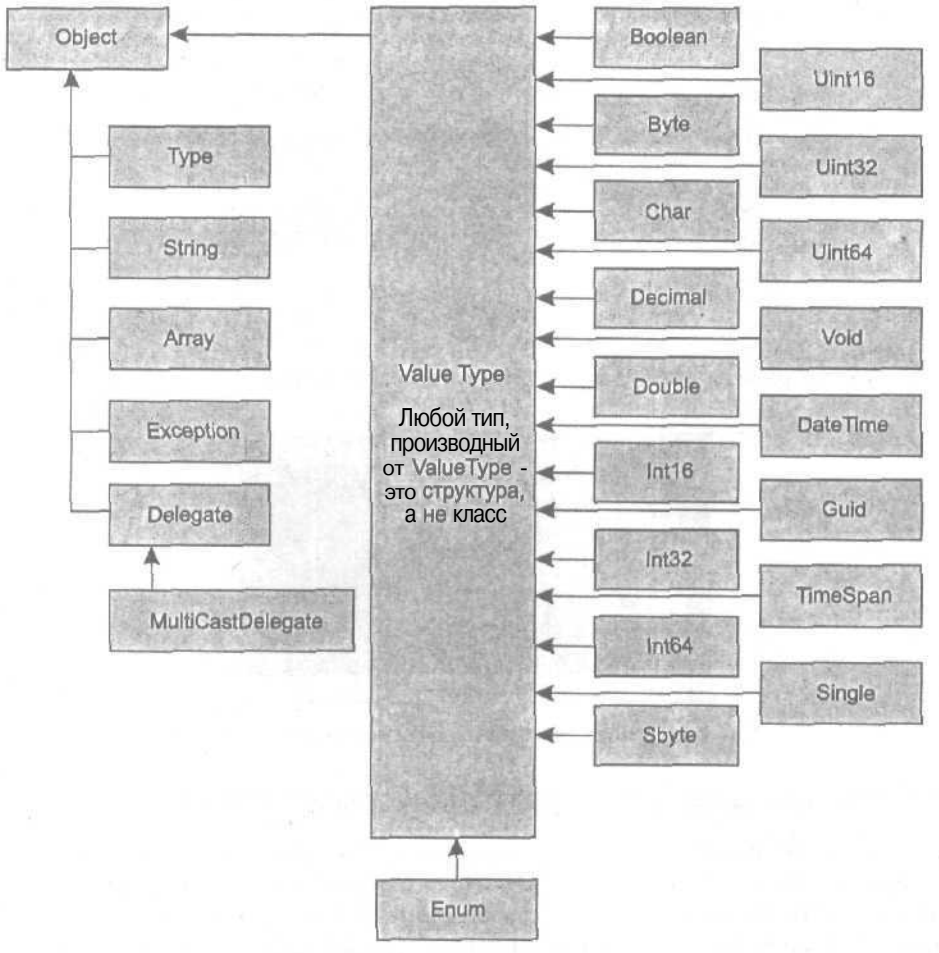
Как мы уже неоднократно убеждались, все типы происходят от System.Object. Поскольку тип данных int — это всего лишь более простая и удобная Запись системного типа Int32, такой синтаксис вполне допустим:

```
// Помните! int в C# - всего лишь псевдоним для System.Int32
Console.WriteLine(12.ToString());
```

Обратите также внимание, что только часть типов C# является CLS-совместимой. Если вы создаете свои типы, которые будут использоваться в многоязыковой среде, желательно ограничиться лишь CLS-совместимыми типами. Запоминать на самом деле надо не так много: достаточно принять за правило не использовать беззнаковые типы в определениях любых открытых членов типов. Этим вы гарантируете, что ваши классы, интерфейсы и структуры смогут нормально работать с любым языком .NET. В главе 7 мы познакомимся со специальным атрибутом уровня сборки, который поможет вам убедиться в том, что все типы данных в сборке соответствуют правилам CLS.

Большинство встроенных типов данных C# являются производными от типа System.ValueType. Единственное назначение этого типа — замещение некоторых методов, определенных в System.Object, таким образом, чтобы они могли нормально работать со структурными типами. В действительности сигнатуры этих методов в System.Object и System.ValueType совпадают. За счет замещения, например, при сравнении значений двух целочисленных переменных используется структурная, а не ссылочная семантика:

```
System.Int32 intA = 1000; // То же. что int intA = 1000:
System.Int32 intB - 1000; // То же. что int intB = 1000;

// Сравнение значений дает "истинно"
if (intA - - intB)
      Console.WriteLine("Same value!");
```



Рис, 2.10. Иерархия системных типов

У каждого системного типа данных (к примеру, Int32, Char, Boolean и т. п.) есть схожий набор членов, которые могут оказаться весьма полезными. Полный список всех этих членов мы приводить не будем (его легко получить с электронной документацией), однако о двух членах все же стоит упомянуть. Речь идет о свойствах MaxValue и MinValue. Первое позволяет получить информацию о максимальном значении, для хранения которого можно использовать данный тип, а второе, соответственно — о минимальном. Предположим, что мы создали переменную типа System.UInt16 и занялись кое-какими операциями с ней:

```
// Конечно же. System.Int16 - это ushort в C#

class MyDataTypes
{
        public static int Ma1n(str1ng[] args)
        {
                // Работаем с UInt16 как со структурным типом
                System.UInt16 myUInt16 = 30000;
```

```
            Console.WriteLine("Max for an UInt16 is: {0}", UInt16.MaxValue);
            Console.WriteLine("Min for an UInt16 is: {0}", UInt16.MinValue);
            Console.WriteLine("Your value is: {0}", myUint16.ToString());
            Console.WriteLine("I am a: {0}", myUint16.GetType().ToString());

            // Почти то же самое, но уже используем псевдоним C# ushort
            // для типа System.Uint16
            ushort myOtherUInt16 = 12000;
            Console.WriteLine("\nYour value is: {0}", myOtherUInt16.ToString());
            Console.WriteLine("I am a: {0}", myOtherUInt16.GetType().ToString());

            return 0;
        }
}
```

Результат работы этой программы представлен на рис. 2.11.

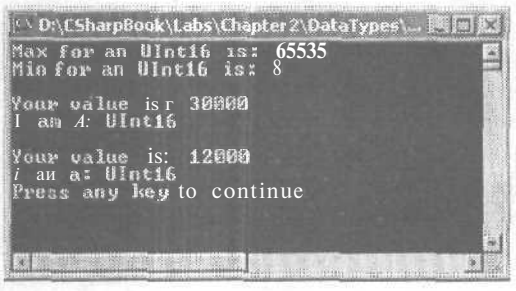


Рис. 2.11. Экспериментируем с системными типами данных

Избранные заметки о некоторых типах данных

Некоторые системные типы данных требуют особого пояснения. Первый из этих типов — `System.Boolean`. В отличие от C и C++ теперь присваивать переменным этого типа действительно можно только два значения — `true` (истинно) и `false` (ложно). Если в C и C++ мы вполне могли присваивать аналогичному логическому типу данных и `0`, и `1`, и `-1`, то C# уже такого не допускает:

```
// Вольница с типом bool закончилась!
bool b = 0;          // Так в C# уже нельзя.
bool b2 = -1;        // Так тоже нельзя.
bool b3 = true;      // А вот так - можно.
bool b4 = false;     // И так можно.
```

Еще одна важная вещь, о которой нельзя не упомянуть, — то, что для текстовых данных в C# теперь используются только два типа данных: `string` и `char`. Думаю, что многие программисты почувствуют невыразимое *облегчение*, узнав, что отпала потребность в таких замечательных типах данных, как `char*`, `wchar_t*`, `LPSTR`, `LPCSTR`, `BSTR` и `OLECHAR`... Многие программисты согласятся со мной, что операции с текстовыми данными в COM и Win32 производились не самым простым способом. В этом отношении .NET, где для работы с текстом предназначены только два типа данных (оба работают с Unicode), — это большой шаг вперед.

Код приложения DataTypes можно найти в подкаталоге Chapter 2.

От структурного типа к ссылочному типу и наоборот: упаковка и распаковка

В C# предусмотрен очень простой механизм для преобразования структурного типа в ссылочный. Он получил название *упаковки* (boxing). Предположим, что у нас есть переменная простого структурного типа данных — short:

```
// Создаем переменную типа short и присваиваем ей значение
short s = 25;
```

А вот как выглядит процесс упаковки — преобразования объекта структурного типа в ссылочный:

```
// Упаковываем переменную s:
object objShort = s;
```

Упаковка — это процесс явного преобразования структурного типа в ссылочный. При этом происходит следующее: в управляемой куче создается новый объект и в него копируются внутренние данные старого объекта из стека (в нашем случае копируется только значение — 25).

Противоположная операция называется *распаковкой* (unboxing). Распаковка — это преобразование ссылки на объект в оперативной памяти обратно в структурный тип. Перед выполнением распаковки среда выполнения производит проверку на совместимость между типом объекта в оперативной памяти и тем структурным типом, в который будет производиться распаковка. Например, следующий код будет выполнен успешно только в том случае, если objShort — это действительно ссылка на объект типа short:

```
// Обратная распаковка объекта
short anotherShort = (short)objShort;
```

Если же среда выполнения обнаружит, что мы пытаемся произвести распаковку в неподходящий тип данных, будет сгенерировано исключение InvalidCastException (про приведение типов — casting и про исключения будет подробнее рассказано в следующей главе):

```
// Неверная распаковка!
public static int Main(string[] args)
{
...
        try
        {
                // Мы пытаемся распаковать в тип данных string объект,
                // исходный тип данных для которого - short!
                string str = (string)objShort;
        }
        catch(InvalidCastException e)
        {
                Console.WriteLine("OOPS!\n{0}", e.ToString());
        }
```

i"

Результат выполнения этой программы представлен на рис. 2.12.

Теперь самое время выяснить следующий вопрос: а когда вообще следует заниматься упаковкой и распаковкой? Ответ таков — не слишком часто. В подавляющем большинстве случаев операции по упаковке и распаковке выполняются компилятором C# полностью автоматически. Например, если вы передаете структурный тип методу, который принимает в качестве параметра объект, упаковка будет произведена безо всякого вашего участия:

```
.// Предположим, что у нас есть такой метод: public static void Foo(object o)
int x = 99;
Foo(x); // Произойдет автоматическая упаковка
```

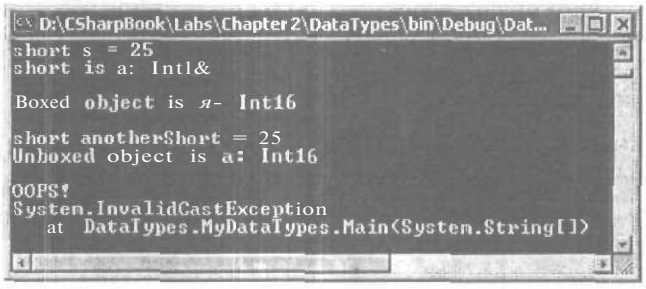


Рис. 2.12. Ошибка при распаковке

Явная упаковка или распаковка используются в C# только тогда, когда это позволяет повысить производительность вашего приложения. Мы еще встретимся с упаковкой и распаковкой ближе к концу этой главы.

Код приложения DataTypes можно найти в подкаталоге Chapter 2.

Значения по умолчанию для встроенных типов данных

Как уже говорилось, у всех встроенных типов данных в мире .NET есть свои значения по умолчанию. Когда вы создаете экземпляр класса, его переменным-членам автоматически присваиваются эти значения. В качестве примера мы воспользуемся следующим определением класса:

```
// В C# при создании класса всем переменным-членам автоматически присваиваются значения
// по умолчанию, что исключает возможность использования неинициализированных переменных
class DefValObject
{
        public sbyte        theSignedByte;
        public byte         theByte;
        public short        theShort;
        public ushort       theUShort;
        public int          theInt;
        public uint         theUInt;
        public long         theLong;
        public ulong        theULong;
        public char         theChar;
        public float        theFloat;
        public double       theDouble;
        public bool         theBool;
```

```
    public decimal          theDecimal;
    public string           theStr;
    public object           theObj;

    public static int Main(string[] args)
    {
        DefValObject v = new DefValObject();
        return 0;      // На этом месте можно установить брейкпойнт и посмотреть
                       // значения в окне Autos
    }
}
```

Если вы перейдете в режим отладки, то вы сможете убедиться, что сразу же после создания экземпляра класса DefValObject всем переменным-членам будут присвоены значения по умолчанию (рис. 2.13).

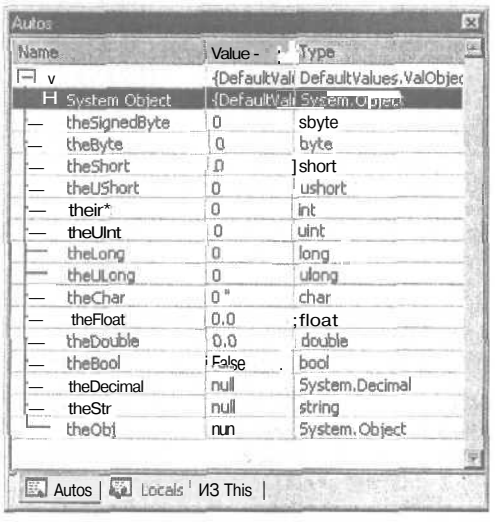


Рис. 2.13. В .NET для всех типов предусмотрены безопасные значения по умолчанию

Однако если вы создаете переменные не внутри класса, а внутри метода, значения по умолчанию для них уже применяться не будут. В этой ситуации вы должны *обязательно* присвоить им начальные значения. Рассмотрим такую ситуацию:

```
// Ошибка компилятора! Для переменной localInt необходимо присвоить начальное значение
public static void MainC)
{
    int localInt;
    Console.WriteLine(localInt.ToString());
}
```

Решение проблемы очевидно:

```
// Все счастливы и довольны.
public static void Main()
{
    int localInt = 0;
    Console.WriteLine(localInt.ToString());
}
```

Из правила, относящегося к обязательному присвоению значений переменным, создаваемых внутри определения метода, есть одно исключение. Если переменная действует как «внешний» параметр — об этом будет рассказано ниже в этой главе, — присваивать начальное значение этой переменной необязательно. В методах, которые определяют такие параметры, подразумевается, что значения таким переменным будут присвоены на уровне вызывающей функции.

Код приложения DefaultValues можно найти в подкаталоге Chapter 2.

Константы

Кроме переменных в программах часто используются их в некотором смысле противоположности — константы. Как и в C, и в C++, в C# константы определяются при помощи ключевого слова const. Константы можно определять и на уровне методов, но гораздо чаще они используются на уровне определений классов. Вот пример:

```
using System;
class MyConstants
{
        // Эти константы определены на уровне класса
        public const int myIntConst = 5;
        public const string myStringConst = "I'm a const";

        public static void Main()
        {
                // Эта константа определена на уровне метода
                const string localConst = "I am a rock, I am an island";

                // А теперь мы применяем константы (обратите внимание на указание
                // диапазона, в которой они были созданы.
                Console.WriteLine("myIntConst = {0}\nmyStringConst - {1}",
                                  MyConstants.myIntConst,
                                  MyConstants.myStringConst);

                Console.WriteLine("Local constant: {0}", localConst);
        }
}
```

Иногда удобно использовать служебный класс, созданный с единственной целью — служить в качестве хранилища констант. При этом нелишне будет позаботиться о том, чтобы пользователь не смог создать экземпляр этого класса:

```
// Чтобы запретить создание экземпляров данного класса, достаточно определить конструктор
как private.
classMyConstants
{
        // Некоторые константы
        public const int myIntConst = 5;
        public const string myStringConst = "I'm a const";

        // А теперь запретим пользователю создавать экземпляры данного класса.
        // ведь единственное назначение этого класса - хранить константы
        private MyConstants(){}
}
```

Запретить создавать экземпляры данного класса можно еще одним способом: определив класс как абстрактный (при помощи ключевого слова abstract). Подробнее о применении этого ключевого слова — в следующей главе, а пока только пример:

```
// Определив класс как абстрактный, мы тем самым запрещаем пользователю создавать
экземпляры этого класса.
abstract class MyConstants
{
        // Некоторые константы
        public const int myIntConst = 5;
        public const string myStringConst = "I'm a const";
}
```

В любом случае при попытке создать экземпляр класса MyConstants компилятор выдаст сообщение об ошибке. Использование специального класса для хранения всех констант представляется особенно полезным в свете того, что C# не позволяет определять глобальные константы.

Последнее, что необходимо добавить: в C# (в отличие от C++) нельзя использовать ключевое слово const как часть объявления метода.

Код приложения Constants можно найти в подкаталоге Chapter 2.

Циклы в C#

В любых языках программирования существуют конструкции, которые позволяют выполнять блок кода до тех пор, пока не будет выполнено определенное условие. Вне зависимости от того, из какого языка программирования вы приходите в мир C#, вы в любом случае обнаружите много знакомого. В C# предусмотрено четыре конструкции для работы с циклами:

- for;
- foreach/in;
- while;
- do/while.

Программисты, использующие C, C++ и Java, без сомнения, знакомы с конструкциями for, while и do/while, однако в этих языках нет конструкции foreach. Программисты, использующие Visual Basic, находятся в более выигрышном положении — им знакома и эта конструкция. Сейчас мы бегло рассмотрим каждую из четырех конструкций C# для работы с циклами.

Выражение for

Если точно известно, сколько раз необходимо выполнить блок кода, конструкция for — это конструкция для чемпионов. По существу мы указываем, сколько раз блок кода должен выполнить сам себя в качестве условия выхода из цикла. Все очень просто:

```
public static int Main(string[] args)
{
        // Обратите внимание! Переменная "i" видима только внутри конструкции "for"
        for(int i - 0: i < 10: i++)
```

```
        {
            Console.WriteLine("Number is: {0}", i);
        }

        // А здесь переменная "i" уже невидима.
        return 0;
}
```

Все старые трюки с циклами из C, C++ и Java будут работать и в C#. Вы можете ставить сложные условия на прерывание цикла, создавать бесконечные циклы и использовать ключевые слова goto, continue и break.

Выражение foreach/in

Программисты, использующие Visual Basic, уже знакомы с преимуществами конструкции For Each. Ее эквивалент в C# позволяет последовательно обрабатывать все элементы массива. Ниже представлен простой пример, в котором конструкция foreach используется для просмотра всех элементов массива символьных строк в поисках совпадений с выражениями COM или .NET.

```
// Обрабатываем все элементы нассива, используя "foreach"
public static int Main(string[] args)
{
        string[] arrBookTitles = new String[] {"Complex Algorithms",
                                               "COM for the Fearful Programmer",
                                               "Do you Remember Classic COM?",
                                               "C# and the .NET platform",
                                               "COM for the Angry Engineer"};
        int COM - 0. NET - 0;

        // Считаем, что книг по COM и .NET (охватывающих оба предмета) пока не существует
        foreach (string s in arrBookTitles)
        {
                if (-1 != s.IndexOf("COM"))
                        COM++;
                else if(-1 != s.IndexOf(".NET"))
                        NET++;
        }
        Console.WriteLine("Found {0} COM references and {1} .NET references.", COM, NET);
        return 0:
}
```

Помимо обработки всех элементов простых массивов, конструкцию foreach можно также использовать для работы с встроенными или пользовательскими коллекциями. В главе 4 мы познакомимся с возможностями этой конструкции для программирования с использованием интерфейсов, а в особенности — при работе со встроенными интерфейсами IEnumerator и IEnumerable.

Выражения while и do/while

Конструкция for — самая удобная в тех ситуациях, когда вы точно знаете, сколько раз необходимо выполнить определенное действие. Если же вам точно неизвестно, сколько раз может потребоваться выполнение блока кода, гораздо удобнее конструкции while и do/while.

Для того чтобы проиллюстрировать работу цикла с этими конструкциями, мы используем возможности C# в отношении работы с файлами (этой теме полностью посвящена глава 11). Для считывания информации из файлов в C# предусмотрен класс StreamReader, определенный внутри пространства имен System.IO. Обратите внимание, что экземпляр класса StreamReader создается как значение, возвращаемое статическим методом File.OpenText(). После открытия файла config.win вы можете обработать каждую его строку (то есть вывести ее на консоль) с использованием метода StreamReader.ReadLine():

```
try     // Это на тот случай, если файла config.win на месте не окажется
{
        // Открываем файл config.win
        StreamReader strReader = File.OpenText("C:\config.win");

        // Считываем каждую строку и выводим ее на консоль
        string strLine;
        while(null != (strLine = strReader.ReadLine()));
        {
              Console.WriteLine(strLine);
        }
        // Файл необходимо закрыть
        strReader.Close();
}
catch(FileNotFoundException e)  // Про исключения рассказано в главе 3
{
        Console.WriteLine(e.Message);
}
```

Конструкция do/while очень похожа на while. Единственное отличие заключается в том, что при использовании do/while мы гарантируем, что соответствующий блок кода будет выполнен по крайней мере один раз (в обычном варианте while это совсем не гарантируется — например, в тех ситуациях, когда условие на продолжение цикла ложно с самого начала). Пример программного кода, в котором используется do/while, представлен ниже.

```
string ans;
do
{
        Console.WriteLine("Are you done? [yes] [no] : ");
        ans = Console.ReadLine();
}while (ans !="yes");
```

Результаты выполнения программ, которые были использованы в качестве примеров в этом разделе, представлены на рис. 2.14.

Код приложения Iterations можно найти в подкаталоге Chapter 2.

Средства управления логикой работы программ в C#

Помимо циклов в любом языке программирования предусмотрены средства для управления логикой работы программ — операторы исходного перехода. В C# для этой цели предусмотрены две простые конструкции — If/else и switch,

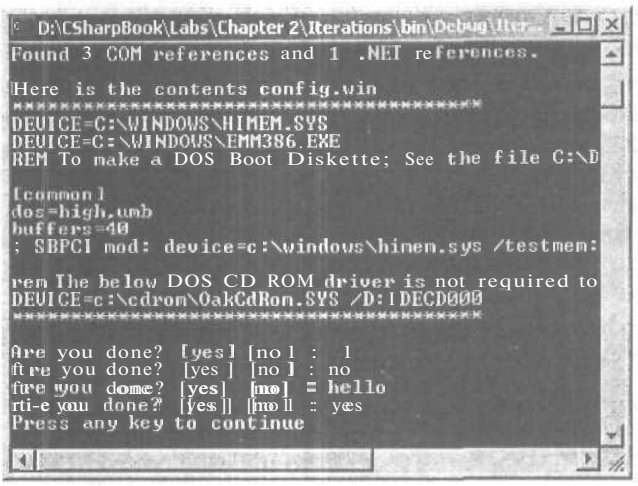


Рис. 2.14. Результаты работы программ с циклами

Конструкция if/else — старинный друг любого программиста. Однако если вы привыкли к особенностям этой конструкции в C и C++, вам необходимо будет освоиться с некоторыми особенностями этой конструкции в C#. В C# конструкция if/else работает только с логическими выражениями (true и false), и поэтому старые трюки, в которых использовались значения 0, -1, и прочие уже не проходят. В C# в конструкции If/else обычно используются операторы, представленные в табл. 2.5.

Таблица 2.5. Операторы сравнения в C#

Оператор C#	Пример использования	Назначение
==	if (age == 30)	Возвращает "true", если два выражения равны
!=	if ("Foo" != myStr)	Возвращает "true", если два выражения не равны
<	if (bonus < 2000)	Возвращает "true", если выражение слева меньше
>	if (bonus > 2000)	Возвращает "true", если выражение слева больше
<=	if (bonus <= 2000)	Возвращает "true", если выражение слева меньше или равно выражению справа
>=	if (bonus >= 2000)	Возвращает "true", если выражение слева больше или равно выражению справа

Программистам, привыкшим к C и C++, необходимо помнить; то, что не равно 0 — не является true для конструкции if/else в C#. Например, такой привычный код приведет к ошибке компилятора C#:

```
// Такой коя не пройдет, так как свойство Length возвращает значение типа int.
// а не bool
string thoughtOfTheDay = "You CAN teach an old dog new tricks";
if (thoughtOfTheDay.Length) // Ошибка!
        {
                // Код на условие
        }
```

Исправить код таким образом, чтобы он нормально работал в C# и выполнял определенное действие тогда, когда длина thoughtOfTheDay не равна нулю, просто — например, вот так:

```
// Теперь проблем не будет
if (0 != thoughtOfTheDay.Length) // Такое выражение разрешается в "true" или "false"
{'
        // Код на условие
!
```

Условия для выражения 1 f могут быть весьма сложными. В них часто используются операторы, представленные в табл. 2.6.

Таблица 2.6. Операторы для условий в C#

Оператор C#	Пример использования	Назначение
&&	if ((age = = 30) && (name = = "Fred"))	Соответствует логическому И (AND)
||	if ((age = = 30) || (name = = "Fred"))	Соответствует логическому ИЛИ (OR)
!	if (!myBool)	Соответствует логическому НЕ (NOT)

Конечно же, в C# есть и выражение else, которое используется для указания кода, который должен быть выполнен, если условие для i f оказалось ложным. Синтаксис конструкции i f/e l se в C# идентичен синтаксису C++ и Java и не очень отличается от синтаксиса, принятого в Visual Basic.

Второе средство для управления логикой выполнения программы в C# — это конструкция switch. Эта конструкция позволяет вам управлять ходом выполнения программы, основываясь на заранее заготовленных вариантах. Предположим, что наша программа должна попросить пользователя ввести одно из трех возможных значений и в зависимости от этого выбрать один из вариантов возможных действий:

```
// Пример применения конструкции "switch"
class Selections
{
        public static int Main(string[] args)
        {
                Console.WriteLine("Welcome to the world of .NET");
                Console.WriteLine("1 = C#\n2 = Managed C++ (MC++)\n3 = VB.NET\n");
                Console.Write("Please select your implementation language:");
                string s = Console.ReadLine();

                // Все встроенные типы данных поддерживают статический метод Parse()
                int n = int.Parse(s);

                switch(n)
                {
                        // C# требует, чтобы каждый вариант (включая "default" -
                        // по умолчанию), в котором предусмотрены команды для выполнения,
                        // содержал выражение выхода "break" или "goto"
                        case 1:
                        Console.WriteLine("Good choice! C# is all about managed
                        code.");
                        break:

                        case 2:
                        Console.WriteLine("Let me guess, maintaining a legacy system?");
```

```
                break;

                case 3:
                Console.WriteLine("VB.NET: It is not just for kids anymore...");
                break;

                default:
                Console.WriteLine("Well...good luck with that!");
                break;
            }
            return 0;
        }
}
```

Результат выполнения этой программы представлен на рис. 2.15.

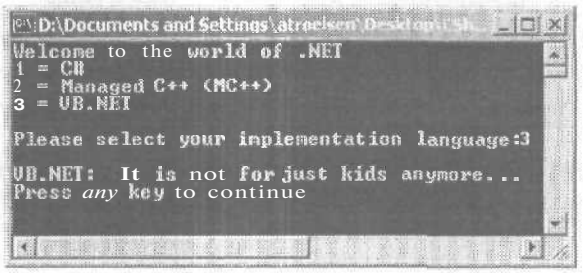


Рис. 2.15. Применение конструкции switch

В конструкции `switch` в C# можно производить выбор, используя в том числе и сравнение со строковыми значениями (не только числовыми). Поддерживается даже значение типа `null` для пустых строк.

Код приложения Selections можно найти в подкаталоге Chapter 2,

Дополнительные операторы C#

Помимо тех операторов, которые были рассмотрены ранее, C# предоставляет в ваше распоряжение множество других операторов. Как правило, все эти операторы ведут себя в C# точно так же, как в C++ и Java. Операторы C# (в порядке убывания приоритета) представлены в табл. 2.7.

Таблица 2.7. Полный набор операторов C#

Категория операторов	Операторы
Унарные	+ — ! ~ ++x x++ --x x--
Операторы умножения и деления	* / %
Операторы сложения и вычитания	+ -
Операторы сдвига	<< >>
Операторы отношения	< > <= >= is as
Операторы равенства	= = !=
Оператор логического И (AND)	&

Категория операторов	Операторы
Оператор логического исключающего ИЛИ (XOR)	^
Оператор логического ИЛИ (OR)	|
Оператор **И (AND) для** проверки условия	&&
Оператор ИЛИ (OR) для проверки **условия**	||
Оператор проверки	?:
Операторы присваивания	= *= /= %= += -= <<= >>= &= ^= |=

Единственные операторы, которые могут быть вам незнакомы, — это операторы is и as. Оператор is используется во время выполнения для проверки совместимости одного типа данных с другим. Как мы увидим в главе 4, этот оператор часто используется, если нужно узнать — поддерживает объект нужный нам интерфейс или нет. Оператор as применяется при приведении типов вниз (подробнее об этом — также в главе 4). На этом мы закончим рассмотрение операторов, подразумевая, что подавляющее большинство из них вам знакомы, а по остальным легко найти необходимую информацию в электронной справке в разделе C# Language Reference.

Определение пользовательских методов класса

Как мы помним, метод — это набор действий, который рассматривается как единое целое и может быть выполнен в ходе работы программы. В C# не существует глобальных методов — любой метод обязательно должен быть членом класса или структуры. Методы могут принимать или не принимать параметры, могут возвращать или не возвращать значения (встроенных или пользовательских типов данных) и могут быть статическими или методами экземпляров.

Модификаторы уровня доступа к методам

В C# для каждого метода существует свой уровень доступа, который определяет, откуда можно будет обратиться к данному методу. Для указания уровня доступа при объявлении метода используются специальные модификаторы. Они перечислены в табл. 2.8.

Таблица 2.8. Модификаторы уровня доступа к методам **в** C#

Модификатор	Назначение
public	Модификатор общедоступности метода
private	Метод будет доступен только из класса, в котором определен данный метод. Если при объявлении метода модификатор явно не указан, по умолчанию используется модификатор private
protected	Метод будет доступен как из класса, в котором он определен, так и из любого производного класса. Для остальных вызовов из внешнего мира этот метод будет недоступен
internal	Метод будет доступен из всех классов внутри сборки, в которой он определен. Из-за пределов этой сборки обратиться к нему будет нельзя
protected internal	Действует как protected или как internal

Подробнее особенности модификаторов protected и Internal будут проанализированы в следующей главе при обсуждении иерархии классов.

Примеры применения модификаторов уровня доступа представлены ниже:

```
// Уровни доступа к методам
class SomeClass
 {
        // Доступен отовсюду
        public void MethodA(){};

        // Доступен только из типов данных SomeClass
        private void MethodB(){};

        // Доступен только из SomeClass и из классов, производных от SomeClass
        // Сна любом нижестоящем уровне иерархии)
        protected void MethodC(){};

        // Доступен только из той же самой сборки
        internal void MethodD(){};

        // Будет действовать как protected или internal
        protected internal void MethodE(){};

        // Будет считаться protected - по умолчанию
        void Method() {};
}
```

Методы, которые объявлены как publ 1c, могут быть доступны напрямую откуда угодно через экземпляр объекта того класса, в котором они определены. К методам, объявленным как private, нельзя обратиться через экземпляр объекта. Такие методы предназначены для вызова из самого этого экземпляра объекта, чтобы помочь ему выполнить какую-либо работу (так называемые частные вспомогательные методы). Например, предположим, что в классе Teenager определены как publ 1c два метода — Complai n() и BeAgreeabl e(), каждый из которых должен выводить символьную строку на системную консоль. Оба этих метода используют частный вспомогательный метод (определенный как private) GetRandomNumber(), который работает с закрытыми переменными-членами типа System. Random, возвращая случайные значения:

```
// Два метода, определенных как public, обращаются к частному вспоногательному методу
using System;
class Teenager
{
        // Тип System.Random нужен для генерации случайных чисел
        private Random r = new Random();

        public string Complain();
        {
                string messages = new string[5]
                        {"Do I have to?",
                         "He started it!",
                         "I'm too tired...",
                         "1 hate school!",
                         "You are sooo wrong."};
                return messages[GetRandomNumber(5)];
        }
```

```
        public string BeAgreeable()
        {
            string[] messages * new string[3]
                    {"Sure! No problem!",
                     "Uh uh.",
                     "I guess so."};
            return messages[GetRandomNumber(3)];
        }

        // Частная вспомогательная функция для получения случайного числа
        private int GetRandomNumber(short upperLimit)
        {
            // Random.Next() возвращает случайное целочисленное число в диапазоне
            // между 0 и upperLimit
            return r.Next(upperLimit);
        }

        public static void Main(string[] args)
        {
            // Майк, можешь начинать жаловаться
            Teenager mike = new Teenager();
            for (int 1 = 0; i < 10: i++)
            {
                Console.WriteLine("mike.Complain());
            }
        }
}
```

Результат выполнения нашей программы представлен на рис. 2.16.

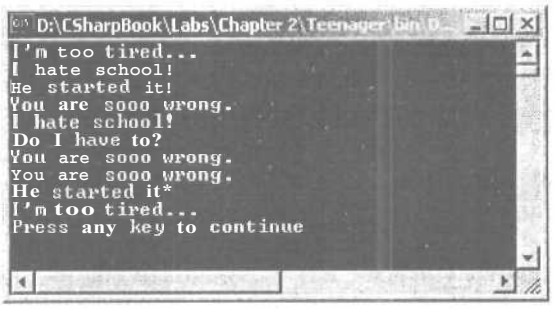


Рис. 2.16. Майк жалуется (использование частного вспомогательного метода)

Преимущество выделения GetRandomNumber() в отдельный частный вспомогательный метод заключается в том, что к этому методу теперь смогут обращаться самые разные части класса Teenager. Альтернатива такому подходу — продублировать программную логику для возвращения случайных чисел в разных местах класса. В нашем случае такое дублирование не причинит неудобств, однако представьте себе ситуацию, в которой GetRandomNumber потребовал бы десятков и сотен строк кода.

Обратимся еще к одному интересному моменту нашей программы — использованию типа System.Random. Очевидно, что этот класс используется для генерации случайных чисел. Метод Random.Next О возвращает число в диапазоне между 0 и указанным верхним предельным значением. Конечно же, в классе Random имеются

и другие полезные члены, однако поскольку для рассказа о них нам придется уклониться от нашей темы, мы можем посоветовать вам обратиться к электронной документации по C#.

Статические методы и методы экземпляров

Как мы уже не один раз видели, методы могут быть объявлены как статические — с использованием ключевого слова static. Однако что это значит? Статический метод (static method) может быть вызван напрямую через уровень класса, без необходимости создавать хотя бы один экземпляр объекта данного класса. По этой причине метод Main() всегда объявляется как static — чтобы этот метод мог начать выполняться еще до создания первого экземпляра класса, в котором он определен.

Чтобы показать использование статических методов на практике, предположим, что мы определили метод Complain() следующим образом:

```
// Молодежь сейчас так часто жалуется, что создавать для этого специальные экземпляры //
// объектов - лишняя работа
public static string Complain()
{
        string[] messages = new string[5]
                {"Do I have to?",
                 "He started it!",
                 "I'm too tired...",
                 "I hate school!",
                 "You are sooo wrong."};
        return messages[GetRandomNumber(5)];
}
```

Вызывать статические методы очень просто: достаточно указать вместе с именем этого метода имя класса, в котором он определен:

```
// Вызываем статический метод Complain() класса Teenager
public static void Main(string[] args)
{
        for int i=0; i < 40; i++)
                Console.WriteLine(Teenager.Complain());
}
```

Методы экземпляров (instance methods), в отличие от статических методов, применяются на уровне экземпляров объектов. Если метод не объявлен как static, то он считается методом экземпляров по умолчанию. Для вызова метода экземпляра необходимо вначале создать объект класса, в котором определен данный метод. Затем метод вызывается через объект данного класса:

```
// Для вызова метода Complain() мы должны создать экземпляр объекта класса Teenager
Teenager joe = new Teenager();
joe.Complain();
```

Код приложения Teenager можно найти в подкаталоге Chapter 2.

Статические данные

Классы в C# могут содержать не только статические методы, но и статические данные — как правило, статические переменные. Чтобы лучше представить себе статические данные, вначале мы обратимся к обычным данным класса. Такие обыч-

ные данные каждый объект данного класса хранит отдельно и независимо от других объектов данного класса. Предположим, что у нас имеется следующий класс:

```
class Foo
{
        public int intFoo;
}
```

Вы можете создать любое количество объектов Foo и присвоить переменной IntFoo в каждом из них индивидуальные значения:

```
// В каждой объекте Foo - своя копия переменной IntFoo
Foo f1 = new Foo();
f1.intFoo = 100;

Foo f2 = new Foo();
f2.intFoo = 993;

Foo f3 - new Foo();
f3.intFoo = 6;
```

В отличие от обычных данных статические данные совместно используются всеми объектами того класса, в котором эти данные были определены. Вместо локальных копий переменной в каждом из объектов существует единственная копия статической переменной, которой пользуются все объекты. Предположим, что у нас есть класс Airplane с единственной статической переменной NumberInTheAir. Пусть значение этой переменной увеличивается на единицу всякий раз, когда у нас срабатывает конструктор нашего класса:

```
// Обратите внимание на применение ключевого слова "static"
class Airplane
{
        // Эта статическая переменная будет совместно использоваться всеми
        // объектами Airplane
        private static int NumberInTheAir = 0;

        public Airplane()
        {
                NumberInTheAir++;
        }

        // Метод для получения значения NumberInTheAir через экземпляр
        // объекта Airplane
        public int GetNumberFromObject() {return NumberInTheAir;}

        // Статический метод - для получения значения NumberInTheAir напрямую
        // через класс Airplane
        public static int GetNumber() {return NumberInTheAir;}
}
```

Обратите внимание на два определенных в классе A1rplane метода. Оба они возвращают текущее количество объектов A1rplane, которое будет создано в нашем приложении. Однако для того, чтобы воспользоваться методом GetNumberFromObject(), нам необходимо создать по крайней мере один экземпляр объекта Airplane, через который будет производиться вызов этого метода. Для того чтобы воспользоваться методом GetNumber(), никаких объектов создавать не нужно, поскольку этот метод определен как статический. Вот пример приложения, в котором используются эти методы:

```
// Экспериментируем со статическими членами
class StaticApp
{
        public static int Main(string[] args)
        {
                // Создадим несколько самолетов
                Airplane a1 = new Airplane();
                Airplane a2 = new AirplaneO;

                // Сколько взлетело самолетов?
                Console.WriteLine("Number of planes: {0}", a1.GetNumberFromObject());

                Console.WriteLine("Number of planes: {0}", Airplane.GetNumber());

                // Добавим самолетов
                Airplane a3 = new Airplane();
                Airplane a4 = new Airplane();

                // А теперь сколько?
                Console.WriteLine("Number of planes: {0}", a3.GetNumberFromObject());

                Console.WriteLine("Number of planes: {0}", Airplane.GetNumber());

                return 0;
        }
}
```

Результат выполнения нашей программы представлен на рис. 2.17.

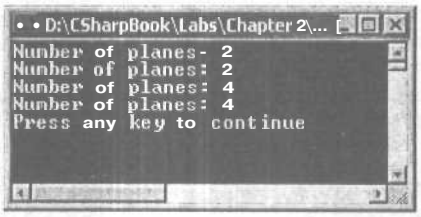


Рис. 2.17. Статические данные совместно используются всеми объектами класса

Как мы видим по результатам работы нашего приложения, статические переменные совместно используются всеми объектами класса, в котором они были определены. В этом и состоит их назначение. Можно сказать, что статические данные предназначены для хранения информации на уровне всего класса, а не отдельных объектов.

Код приложения StaticTypes можно найти в подкаталоге Chapter 2.

Интересное рядом: некоторые статические члены класса Environment

Environment — это еще один из множества классов, определенных в пространстве имен System. В этом классе определено несколько статических типов, которые можно использовать для получения полезной информации об операционной системе, в которой будет работать ваше приложение, а также о самой среде выполнения .NET. Рассмотрим применение некоторых из этих статических типов на примере:

```
// Вот несколько Сно далеко не все) полезных статических типов класса Environment
using System;
class Environment;
{
        public static int Main(string[] args)
        {
                // Под какой операционной системой мы работаем?
                Console.WriteLine("Current OS: {0}", Environment.OSVersion);

                // Текущий каталог?
                Console.WriteLine("Current Directory: {0}", Environment.CurrentDirectory);

                // А теперь выведен список логических дисков:
                string[] drives = Environment.GetLogicalDrives();

                for(int i - 0; i < drives.Length; 1++)
                        Console.WriteLine("Drive {0} : {1}", i, drives[i]);

                // А какая версия платформы .NET у нас используется?
                Console.WriteLine("CurrentVeersion of .NET: {0}", Environment.Version);

                return 0;
        }
}
```

Результат работы этой программы, представлен на рис. 2.18.

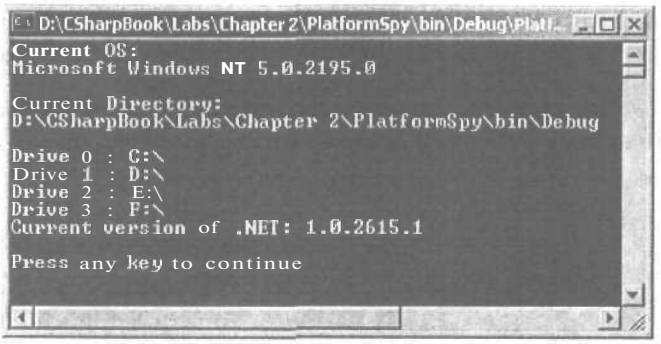


Рис. 2.18. Работа с переменными окружения в C#

Код приложения PlatformSpy можно найти в подкаталоге Chapter 2.

Модификаторы для параметров методов

В большинстве случаев методы используют параметры. Если у вас есть опыт работы с COM, вы, безусловно, знакомы с атрибутами [in], [out] и [in, out] в IDL. Классические объекты COM используют эти атрибуты для того, чтобы однозначно определить направление передачи данных (и решить связанные с этим вопросы выделения памяти) для параметров метода, связанных с указанным интерфейсом. Несмотря на то что в C# IDL не используется, однако атрибуты для параметров методов (они называются модификаторами параметров) остались. Перечень этих модификаторов представлен в табл. 2.9.

Таблица 2.9. Модификаторы параметров C#

Модификатор	Назначение
(нет)	Если параметр никак не помечен, то по умолчанию считается, что этот параметр — входящий (для передачи методу), передаваемый как значение. Вместо пропуска модификатора можно использовать модификатор in — результат будет тем же самым
out	Аналогично атрибуту [out] в IDL: исходящий параметр, который возвращается вызванным методом вызывающему
ref	Аналогично атрибуту [in, out] в IDL: исходное значение присваивается вызывающим методом, но оно может быть изменено вызванным. При этом происходит передача параметра по ссылке
params	Этот модификатор позволяет передавать целый набор параметров как единое целое. В любом методе может быть только один модификатор params, и параметр с этим модификатором может быть только последним параметром в списке параметров метода. Можно сказать, что это — примерный аналог SAFEARRAY в COM

Вначалемыпроиллюстрируемнапримерепримененияmодификаторовin— подразумеваемое, и out — явное. Ниже приведена версия метода Add(), который возвращает сумму двух целочисленных чисел C#, используя ключевое слово out:

```
// Значения исходящим - out - параметрам присваиваются вызываемым методом
public void Add(int x. int y, out int ans)
{
        ans = x + y:
}
```

Вызов метода с исходящими параметрами также требует применения ключевого слова out. Например:

```
// Предположим, что метод Add() определен в классе с именем Methods
public static void Main()
{
...
        Methods m = new Methods();
        int  ans;       // Исходное значение присваивать не обязательно - значение вскоре
                        // присвоит вызываемый метод

        // Обратите внимание на использование ключевого слова out в синтаксисе
        // вызова метода
        m.Add(90. 90, out ans);
        Console.WriteLine("90 + 90 = {0}", ans);
}
```

Как можно убедиться, эта версия метода AddO работает так же, как и классическая:

```
public int Add (int x. int y)
{
        return x + y;
}
```

Параметры, передаваемые по ссылке (с использованием модификатора ref), обычно применяются для того, чтобы метод внес изменения в существующие значения. В качестве примера можно привести процедуры сортировки. Обратите вни-

мание на различия между исходящими (out) параметрами и параметрами, передаваемыми по ссылке (ref):

- Исходящие параметры перед передачей вызываемому методу инициализировать не обязательно. Причина очевидна — о присвоении значения этому параметру должен позаботиться вызываемый метод.
- Параметры, передаваемые по ссылке, обязательно должны быть инициализированы перед передачей вызываемому методу. Причина также очевидна: вы передаете ссылку на существующий тип. Если вы еще не присвоили начальное значение, то это все равно, что передавать пустой указатель!

Проиллюстрируем применение ключевого слова ref на примере:

```
// Метод принимает параметр, передаваемый по ссылке
public void UpperCaseThisString(ref string s)
{
        // Возвращаем символьную строку, в которой все буквы станут заглавными
        s = s.ToUpper();
}

// Вызываем этот метод через Main()
public static void Main()
{
        ...
        // Используем ключевое слово ref
        string s = "Can you really have sonic hearing for $19.00?";
        Console.WriteLine("Before: {0}", s);

        m.UpperCaseThisString(ref s)
        Console.WriteLine("After: {0}", s);
}
```

Последний модификатор, который мы рассмотрели, — это модификатор params. Этот модификатор используется для передачи набора параметров (в неопределенном количестве) как одного параметра. Для того чтобы вы не запутались, представляя себе этот механизм, разберем его на примере. Представим, что в нашем распоряжении есть следующий метод:

```
// Метол принимает два физических параметра
public void DisplayArrayOfInts(string msg, params int[] list)
{
        Console.WriteLine(msg);

        for (int i=0; i < list.Length; i++)
                Console.WriteLine(list[i]);
}
```

Этот метод объявлен таким образом, что он принимает только два физических параметра: один — типа string, а второй — как параметризованный массив значений типа int. Можно считать, что этот метод предлагает вам: «Дайте мне символьную строку как первый параметр и *любое количество* целых чисел как второй». Вы можете вызвать этот метод любым из следующих способов:

```
// Передаем значения методу, объявленному с модификатором params
1nt[] intArray = new int[3] {10,11,12};
m.DisplayArrayOfInts ("Here is an array of ints", intArray);
m.DisplayArrayOfInts ("Enjoy these 3 ints", 1, 2, 3);
m.DisplayArrayOfInts ("Take some more!", 55. 4, 983. 10432, 98, 33);
```

Изучив приведенный код, можно заметить, что выделенные элементы в любом из вариантов соответствуют второму параметру метода DisplayArrayOfInts. Конечно же, вы не обязаны использовать с ключевым словом params исключительно простейшие целочисленные значения. Предположим, что класс Person в этот раз определен следующим образом:

```
// Новая разновидность класса Person
class Person
{
        private string fullName;
        private byte age;

        public Person(string n, byte a)
        {
                fullName = n;
                age = a;
        }

        public void PrintInfo()
        {
                Console.WriteLine("{0} is {1} years old", fullName, age);
        }
}
```

Теперь предположим, что в классе Methods определен еще один метод, использующий ключевое слово params. Однако на этот раз мы собираемся принимать не массив целочисленных значений, а массив объектов — то есть *массив, состоящий из чего угодно.*

Давайте воспользуемся этой возможностью, применим ее к классу Person. Техническое задание будет такое: пусть наш метод принимает массив объектов, проверяет все объекты, и если среди них обнаруживаются объекты типа Person, то вызывается метод PrintInfo() для этого объекта. Объекты любого другого типа просто выводятся на системную консоль.

```
// Что же мне пришлют на этот раз?
public void DisplayArrayOfObjects(params object[] list)
{
        for (int i=0; i < list.Length; i++)
        {
                if(list[i] is Person)          // Является ли текущий объект в массиве
                                               // объектом класса Person?
                {
                        ((Person)list[i]).PrintInfo();  // Если да, вызываем нужный метод
                }
                else
                        Console.WriteLine(list[i]);
        }
        Console.WriteLine();
}
```

Вызов этого метода может выглядеть следующим образом:

```
// Вначале создаем объект класса Person
Person p = new Person("Fred", 93);
m.DisplayArrrayOfObjects(777, p, "I really am an instance of System.String");
```

Результат работы программы с вызовом данного метода представлен на рис. 2.19.

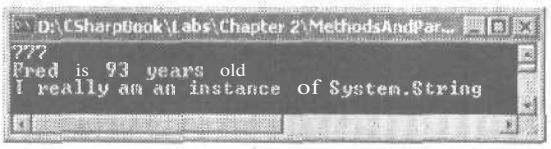


Рис. 2.19. Возможности модификатора params

Как мы могли убедиться, C# предлагает множество средств для удобной и эффективной работы с параметрами. Отдельно заметим для программистов на C и C++: в C# вы можете привязывать параметры, передаваемые методам, к указателям (используя ключевое слово ref) без необходимости использовать не самые изящные конструкции с операторами * и &.

Код приложения MethodsAndParams можно найти в подкаталоге Chapter 2.

Работа с массивами

Массивы в C# с точки зрения синтаксиса практически не отличаются от массивов в C, C++ и Java. Однако если посмотреть на их внутреннее устройство, массив C# — это тип, производный от класса System.Array. Поэтому все массивы C# обладают общим набором членов.

Формальное определение массива выглядит так: массив — это набор элементов, доступ к которым производится с помощью числового индекса. Индексы могут содержать элементы любого встроенного типа данных C#, в том числе могут существовать массивы объектов, интерфейсов и структур. В C# массивы могут быть простыми или многомерными. Массивы объявляются путем помещения квадратных скобок ([ ]) после указания типа данных для элементов этого массива, например:

```
// Массив символьных строк с 10 элементами {0, 1, . . . , 9}
string[] booksOnCOM;
booksOnCOM = new string[10];

// Массив символьных строк с 2 элементами {0, 1}
string[] booksOnPL1 = new string[2];

// Массив символьных строк из 100 элементов {0, 1, . . . , 99}
string[] booksOnDotNet = new string[100];
```

Как мы видим, в первом варианте на объявление массива нам потребовалось две строки. Однако те же самые действия мы можем сделать и в одной строке (это верно и для любых других объектов C#). Однако в любом случае для создания массива фиксированного размера мы должны использовать ключевое слово new. Такое объявление массива приведет к ошибке компилятора:

```
// При определении массива фиксированного размера мы обязаны использовать
// ключевое слово new
int[4] ages - {30, 54. 4, 10}; ,    // Ошибка!
```

Помните, что размер массива задается при его создании, но не объявлении. Если мы объявляем массив фиксированного начального размера, мы обязаны использовать ключевое слово new. Но если мы возлагаем обязанность по определению необходимого размера массива на компилятор, можно обойтись и без new:

```
// Будет автоматически создан массив с 4 элементами. Обратите внимание на отсутствие
ключевого слова "new" и на // пустые квадратные скобки
1nt[] ages - {20. 22. 23. 0};
```

Как и во многих других языках, в C# можно производить заполнение массива, перечисляя элементы последовательно в фигурных скобках, а можно использовать для этой цели числовые индексы. Результат применения обоих вариантов будет совершенно одинаковым:

```
// Используем последовательное перечисление элементов массива:
string[] firstNames - new string[5] {"Steve", "Gina", "Swallow", "Baldy", "Gunner"};

// Используем числовые индексы:
str1ng[] firstNames = new string[5];
firstNames[0] - "Steve";
firstNames[1] - "Gina";
firstNames[2] = "Swallow";
firstNames[3] - "Baldy";
firstNames[4] - "Gunner";
```

Важное различие между массивами C++ и C#, о котором необходимо упомянуть, заключается в том, что в C# элементам массива автоматически присваиваются значения по умолчанию в зависимости от используемого для них типа данных. Например, для массива целых чисел всем элементам будет изначально присвоено значение 0, для массива объектов — значение NULL и т. д.

Многомерные массивы

Помимо простых массивов — массивов с одним измерением в C# поддерживаются также две основные разновидности многомерных массивов. Первую разновидность многомерных массивов иногда называют «прямоугольным массивом». Такой тип массива образуется простым сложением нескольких измерений. При этом все строки и столбцы в данном массиве будут одинаковой длины. Объявление и заполнение прямоугольного многомерного массива производится следующим образом:

```
// Прямоугольный многомерный массив
int[.] myMatrix;
myMatrix = new int[6. 6];

// Заполняем массив б на б:
for (int i = 0; i < 6; i++)
        for (int j = 0; j < 6; j++)
                myMatrix[i. j] = i*j;

// Выводим элементы многомерного массива на системную консоль
for (int i = 0; i < б; i++)
{
        for (int j = 0; j < 6; j++)
        {
                Console.Write(myMatrix[i. j] + "\t");
        }
        Console.WriteLine();
}
```

Результат выполнения данной программы представлен на рис. 2.20. При этом прямоугольная сущность этого типа многомерного массива видна очень хорошо.

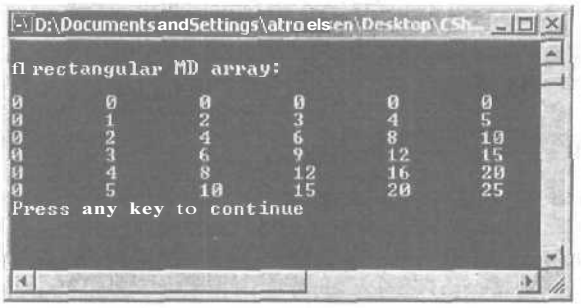


Рис. 2.20. Многомерный прямоугольный массив

Второй тип многомерного массива можно назвать «ломаным» (jagged). Такой массив содержит в качестве внутренних элементов некоторое количество внутренних массивов, каждый из которых может иметь свой внутренний уникальный размер. Например:

```
// "Ломаный" многомерный массив (массив из массивов). В нашем случае - это массив
// из пяти внутренних массивов разного размера
int[][] myJagArray = new int[5][];

// Создаем "ломаный" массив
for (int i = 0; i < myJagArray.Length; i++)
{
      myJagArray[i] = new int[i + 7];
}

// Выводим каждую строку на системную консоль (как мы помним, каждому элементу
// присваивается значение по умолчанию - в нашем случае 0)
for (int i = 0; i < 5; i++)
{
      Console.Write("Length of row {0} is {1}:\t", i, myJagArray[i].Length);
      for (int j = 0; j < myJagArray[i].Length; j++)
      {
            Console.Write(myJagArray[i][j] + " ");
      }
      Console.WriteLine();
}
```

Результат работы этой программы представлен на рис. 2.21 (обратите внимание на разную длину строк этого массива).

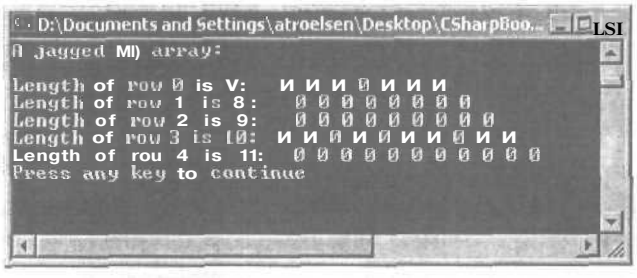


Рис. 2.21. «Ломаный» массив

Теперь, когда мы умеем создавать и заполнять массивы C#, мы можем обратить внимание на главный класс, который является базовым для всех массивов C# — класс System.Array.

Базовый класс System.Array

Все наиболее важные различия между массивами в C++ и C# происходят оттого, что в C# все массивы являются производными от базового класса System.Array. За счет этого любой массив в C# наследует большое количество полезных методов и свойств, которые сильно упрощают работу программиста. Самые интересные методы и свойства (но, конечно, далеко не все) приведены в табл. 2.10.

Таблица 2.10. Некоторые члены класса System.Array

Член класса	Назначение
BinarySearch()	Этот статический метод можно использовать только тогда, когда массив реализует интерфейс IComparer (подробнее — в главе 4). Если этот интерфейс реализован, метод BinarySearch() позволяет найти элемент массива
Clear()	Этот статический метод позволяет очистить указанный диапазон элементов (числовые элементы приобретут значения 0, а ссылки на объекты — null)
CopyTo()	Используется для копирования элементов из исходного массива в массив назначения
GetEnumerator()	Возвращает интерфейс IEnumerator для указанного массива. 06 интерфейсах подробнее будет рассказано в главе 4, сейчас же просто заметим, что этот интерфейс необходим для применения конструкции foreach
GetLength() Length	Метод GetLength() используется для определения количества элементов в указанном измерении массива. Length — это свойство только для чтения, с помощью которого можно получить количество элементов массива
GetLowerBound() GetUpperBound()	Эти методы используются для определения нижней и верхней границы выбранного вами измерения массива
GetValue() SetValue()	Возвращает или устанавливает значение указанного индекса для массива. Этот метод перегружен для нормальной работы как с одномерными, так и с многомерными массивами
Reverse()	Этот статический метод позволяет расставить элементы одномерного массива в обратном порядке
Sort()	Сортирует одномерный массив встроенных типов данных. Если элементы массива поддерживают интерфейс IComparer, то с помощью этого метода вы сможете производить сортировку и ваших пользовательских типов данных (опять-таки за более подробными объяснениями мы отсылаем вас к главе 4)

Рассмотрим некоторые члены System.Array в действии. В нашем примере с помощью статических методов Reverse() и Clear(), а также свойства Length производится вывод на системную консоль информации о массиве firstName:

```
// Создаем несколько массивов символьных строк и экспериментируем
// с членами System.Array
```

```
class Arrays
{
        public static int Main(string[] args)
        {
                // Массив символьных строк
                string[] firstNames - new string[5]
                        {"Steve",
                         "Gina",
                         "Swallow",
                         "Baldy",
                         "Gunner"}
                // Выводим имена в соответствии с порядком элементов в массиве
                Console.WriteLine("Here is the array:");

                for (int i - 0: 1 < firstNames.Length; i++)
                        Console.Write(firstNames[i] + "\t");

                // Расставляем элементы в обратном порядке при помощи статического
                // метода Reverse()
                Array.Reverse(firstNames);

                //  ...и снова выводим имена
                Console.WriteLine("Here is the array once reversed:");
                for (int i = 0; i < firstNames.Length; i++)
                        Console.Write(firstNames[i] + "\t");

                // А теперь вычищаем всех, кроне юного Гуннара
                Console.WriteLine("Cleared out all but one...");
                Array.Clear(firstNames, 1, 4);

                for (int i = 0; 1 < firstNames.Length; i++)
                {
                        Console.Write(firstNames[i] + "\t\n");
                }
                return 0;
        }
}
```

Результаты работы программы представлены на рис. 2.22.

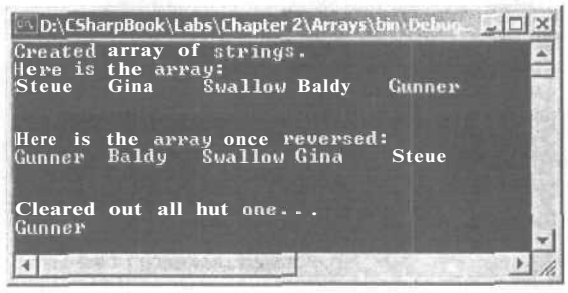


Рис. 2.22. Возможности System.Array

Возможность рассматривать массивы как объекты — это удобство раньше смогли оценить программисты, использующие Java. С появлением C# и платформы .NET те же самые преимущества получают и обычные Windows-программисты.

Закончим наше повествование о массивах на еще одной радостной ноте: теперь вы можете забыть об ужасах при работе со структурой COM SAFEARRAY (по крайней мере, после прочтения главы 12).

Код приложения Arrays можно найти в подкаталоге Chapter 2.

Работа со строками

Как мы уже могли убедиться, тип данных string (строки Unicode) — это встроенный тип данных C#. Как и все встроенные типы данных, все строки в мире C# и .NET происходят от единственного базового класса — System.String. Этот базовый класс обеспечивает множество методов, которые призваны выполнить за вас всю черновую работу: возвратить количество символов в строке, найти подстроки, преобразовать все символы в строчные или прописные и т. д. Самые интересные, с нашей точки зрения, члены класса System.String представлены в табл. 2.11.

Таблица 2.11. Некоторые члены класса *System.String*

Имя члена	Назначение
Length	Это свойство возвращает длину указанной строки
Concat()	Этот статический метод класса String возвращает новую строку, «склеенную» из двух исходных
CompareTo()	Сравнивает две строки
Copy()	Этот статический метод создает новую копию существующей строки
Format()	Используется для форматирования строки с использованием других примитивов (числовых данных, других строк) и подстановочных выражений вида {0}, рассмотренных ранее в этой главе
Insert()	Используется для вставки строки внутрь существующей
PadLeft() PadRight()	Эти методы позволяют заполнить («набить») строку указанными символами
Remove() Replace()	Эти методы позволяют создать копию строки с внесенными изменениями (удаленными или замененными символами)
ToUpper() ToLower()	Эти методы используются для получения копии строки, в которой все символы станут строчными или прописными

Обратите внимание на следующие моменты, связанные с работой со строковыми данными в C#. Во-первых, несмотря на то, что тип данных string — это ссылочный тип данных, при использовании операторов равенства (= = и ! =) происходит сравнение *значений* строковых объектов, а не адресов этих объектов в оперативной памяти. Во-вторых, оператор сложения (+) в C# перегружен таким образом, что при применении к строковым объектам он вызывает метод Concat():

```
// = = и != используются для сравнений значений строковых объектов.
// + используется для операции конкатенации.
public static int Main(string[] args)
\
        System.String strObj = "This is a TEST":
        string s = "This is another TEST";

        // Производим сравнение значений строк
```

```
        if(s = = strObj)
              Console.WriteLine("Same info...");
        else
              Console.WriteLine("Not the same info...");

        // А теперь - операция конкатенации
        string newString = s + strObj;
        Console.WriteLine("s + strObj - {0}", newString);

        // А еще System.String предоставляет в ваше распоряжение индексатор
        // для доступа к любому символу массива
        for (int k = 0; k < s.Length; k++)
              Console.WriteLine("Char (0} is {1}", k. s[k]);

        return 0;
  }
```

При запуске этой программы результат должен получиться следующим. Проверка на идентичность строк, конечно же, сообщит, что строки разные. Содержимое newString должно выглядеть так: This is another TESTThis is a TEST. Вы сможете обратиться к любому символу строки с помощью оператора индекса ([]).

Управляющие последовательности и вывод служебных символов

В C#, как и в C, ив C++, и в Java строки могут содержать любое количество управляющих последовательностей (escape characters):

```
// Применение управляющих последовательностей - \t, \\, \n и прочих
string anotherString;
anotherString = "Every programming book need \"HelloWorld\"";
Console.WriteLine("\t" + anotherString);

anotherString = "c:\\CSharpProjects\\Strings\\string.cs";
Console.WriteLine("\t" + anotherString);
```

Если вы уже не можете вспомнить, для чего используется каждая из управляющих последовательностей, обратитесь к табл. 2.12.

Таблица 2.12, Управляющие последовательности

Управляющие последовательности	Назначение
V	Вставить одинарную кавычку в строку
\"	Вставить двойную кавычку в строку
\\	Вставить в строку обратный слэш. Особенно полезно при работе с путями в файловой системе
\a	Запустить системное оповещение (Alert)
\Ь	Вернуться на одну позицию (Backspace)
\f	Начать следующую страницу (Form feed)
\n	Вставить новую строку (New line)
\r	Вставить возврат каретки (carnage Return)

продолжение

Таблица 2.12 (*продолжение*)

Управляющие последовательности	Назначение
\t	Вставить горизонтальный символ табуляции (horizontal Tab)
\u	Вставить символ Unicode
\v	Вставить вертикальный символ табуляции (Vertical tab)
\0	Представляет пустой символ (NULL)

Помимо управляющих последовательностей, в C# предусмотрен также специальный префикс @ для дословного вывода строк вне зависимости от наличия в них управляющих последовательностей. Строки с этим префиксом называются «дословными» (verbatim strings). Это — очень удобное средство во многих ситуациях:

```
// Префикс @ отключает обработку управляющих последовательностей
string finalString = @"\n\tString file: 'C:\CSharpProjects\Strings\string.cs'";
Console.WriteLine(finalString);
```

Результаты работы сразу нескольких наших программ, использующих строки, представлены на рис. 2.23. Обратите внимание на нижнюю строчку: как можно видеть, применение префикса @ предоставляет возможность спокойно производить вывод любого количества служебных символов.

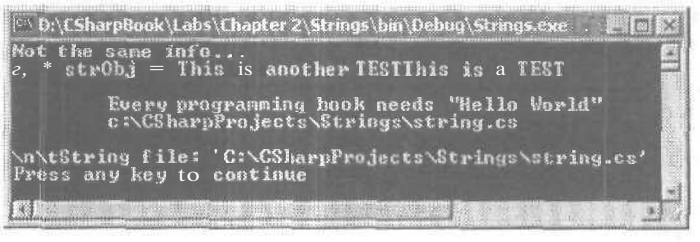


Рис. 2.23. Возможности System.String

Применение System.Text.StringBuilder

При работе со строками в C# необходимо помнить очень важную вещь: значение строки не может быть изменено. Как и в Java, в C# строки являются неизменяемыми (immutable). System.String предоставляет в ваше распоряжение множество методов, однако эти методы только на первый взгляд могут изменять строки. На самом деле они возвращают лишь измененную *копию* строки. Например, при применении метода ToUpper() вы не изменяете сам строковый объект, а всего лишь возвращаете его копию, в которой все буквы станут прописными:

```
// Вносим изменения в строку? На самом деле нет...
System.String strFixed = "This is how I began life";
Console.WriteLine(strFixed);

string upperVersion = strFixed.ToUpper(); // Возвращает "прописную" копию strFixed

Console.WriteLine(strFixed);
Console.WriteLine(upperVersion);
```

Работа с копиями копий строк может в конце концов надоесть. Но C# предлагает еще одно средство, которое позволяет изменять строки напрямую, без создания лишних копии. Это средство — класс StringBuilder, определенный в пространстве имен System.Text. Этот класс во многом напоминает CString в MFC или CComBSTR в ATL. Все изменения, которые вы вносите в объект этого класса, немедленно в нем отражаются, что во многих ситуациях гораздо эффективнее, чем работать с множеством копий.

```
// Демонстрирует применение класса StringBuilder
using System;
using System.Text;    // Здесь живет StringBuilder!

class StringApp
{
        public static int Main(string[] args)
        {
                // Создаем объект StringBuilder и изменяем его содержимое
                StringBuilder myBuffer = new StringBuilder("I am a buffer");
                myBuffer.Append(" that just got longer...");
                Console.WriteLine(myBuffer);
                return 0;
        }
}
```

Помимо добавления класс StringBuilder допускает и другие операции, например удаление определенных символов или их замену. После того как вы добились нужного вам результата, часто бывает удобным вызвать метод ToString(), чтобы перевести содержимое объекта StringBuilder в обычный тип данных String:

```
// Продолжаем демонстрировать применение класса StringBuilder
using System;
using System.Text;   // Здесь живет StringBuilder

class StringApp
{
        public static int Main(string[] args)
        {
                StringBuilder myBuffer = new StringBuilder("I am a buffer");

                myBuffer.Append(" that just got longer...");
                Console.WriteLine(myBuffer);

                myBuffer.Append("and even longer.");
                Console.WriteLine(myBuffer);

                // Делаем все буквы прописными
                string theReallyFinalString = myBuffer.ToString().ToUpper();
                Console.WriteLine(theReallyFinalString);
                return 0;
        }
}
```

Как, наверное, вы уже догадываетесь, класс StringBuilder содержит множество других методов и свойств. Советуем вам познакомиться с ними самостоятельно.

Код приложения Strings можно найти в подкаталоге Chapter 2.

Перечисления C#

Часто бывает удобным создать набор значимых имен, которые будут представлять числовые значения. Представим себе, что вы создаете систему расчета заработной платы для сотрудников. Скорее всего, вам покажется более удобным вместо недружелюбных значений {0, 1. 2, 3} использовать понятные имена VP, Manager, Grunt и Contractor. Конечно же, C#, как и C/C++, предоставляет вам возможность использовать для этой цели перечисления (enumerations). Создание перечисления для наших целей может выглядеть так:

```
// Создаем перечисление
enum EmpType
{
        Manager,        // - 0
        Grunt,          // = 1
        Contractor.     // = 2
        VP              // - 3
}
```

Перечисление EmpType определяет четыре именованные константы, каждой из которых соответствует определенное числовое значение. По умолчанию в перечислениях C# первому элементу соответствует числовое значение 0, второму — n+1, третьему — n+2 и т. д. При необходимости вы легко сможете изменить исходную точку отсчета:

```
// Начинаем со 102
enum  EmpType
{
        Manager  = 102.
        Grunt,          // - 103
        Contractor.     // = 104
        VP              // = 105
}
```

Для элементов перечисления вовсе не обязательно использовать строго последовательные числовые значения. Например, перечисление можно создать так, как показано ниже, и компилятор совсем не будет против:

```
// Элементы перечисления могут иметь произвольные числовые значения
enum EmpType
{
        Manager  = 10.
        Grunt - 1.
        Contractor - 100.
        VP = 99
}
```

Если посмотреть, что происходит с элементами перечисления при компиляции, то окажется, что компилятор попросту подставляет вместо них соответствующие числовые значения. По умолчанию для этих числовых значений компилятор использует тип данных int. Однако вам ничего не мешает явным образом объявить компилятору, что следует использовать другой тип данных, в нашем примере — byte:

```
// Вместо элементов перечисления будут подставляться числовые значения типа byte
enum EmpType : byte
```

```
{
        Manager = 10.
        Grunt = 1.
        Contractor = 100.
        VP = 9
}
```

Точно таким же образом можно использовать любой из основных целочисленных типов данных C#(byte, sbyte, short, ushort, int, uint, long, ulong).

После того как перечисление вместе с необходимым типом данных определено, вы можете использовать его элементы в программе, например, так, как показано ниже. Обратите внимание на то, что в нашем классе определена статическая функция, принимающая в качестве единственного параметра EmpType:

```
using System;
class EnumClass
{
        public static void AskForBonus(EmpType e)
        {
                switch(e)
                {
                case EmpType.Contractor:
                        Console.WriteLine("You are already get enough cash...");
                break;

                case EmpType.Grunt:
                        Console.WriteLine("You havve got to be kidding...");
                break;

                case EmpType.Manager:
                        Console.WriteLine("How about stock options instead?");
                break;

                case EmpType.VP:
                        Console.WriteLine("VERY GOOD, Sir!");
                break;

                default: break;
                }
        }

        public static int Main(string[] args)
        {
                // Создаем тип Contractor
                EmpType fred;
                fred = EmpType.Contractor;

                AskForBonus(fred);

                return 0;
        }
}
```

Базовый класс System.Enum

Все перечисления в C# происходят от единого базового класса System.Enum. Конечно же, в этом базовом классе предусмотрены методы, которые могут существенно

облегчить вашу работу с перечислениями. Первый метод, о котором необходимо упомянуть, — это статический метод GetUnderlyingType(), который позволяет получить информацию о том, какой тип данных используется для представления числовых значений элементов перечисления:

```
// Получаем тип числовых данных перечисления (в нашем примере это будет System.Byte)
Console.WriteLine(Enum.GetUnderlyingType(typeof(EmpType)));
```

Кроме того, вы можете получать значимые имена элементов перечисления по их числовым значениям. Эту работу за вас выполняет статический метод Enum.Format(). В нашем примере переменной типа EmpType соответствовало имя элемента перечисления Contractor (то есть эта переменная разрешалась в числовое значение 100). Для того чтобы узнать, какому элементу переменной соответствует это числовое значение, необходимо вызвать метод Enum.Format, указать тип перечисления, числовое значение (в нашем случае через переменную) и флаг форматирования (в нашем случае — G, что означает вывести как тип string, можно использовать также флаги x — шестнадцатеричное значение и d — десятичное значение):

```
// Этот код должен вывести на системную консоль строку "You are a Contractor
EmpType fred;
fred = EmpType.Contractor;

Console.WriteLine("You are a {0}", Enum.Format(typeof(EmpType), fred, "G"));
```

В System.Enum предусмотрен еще один полезный статический метод — GetValues(). Этот метод возвращает экземпляр System.Array, при этом каждому элементу массива будет соответствовать член указанного перечисления:

```
// Получаем информацию о количестве элементов в перечислении
Array obj = Enum.GetValues(typeof(EmpType));

Console.WriteLine("This enum has {0} members.", obj.Length);

// А теперь выводим имена элементов перечисления в формате string и соответствующие им
числовые значения
foreach(EmpType e in obj)
{
        Console.Write("String name: {0}", Enum.Format(typeof(EmpType), e, "G"));
        Console.Write(" ({0})", Enum.Format(typeof(EmpType), e, "D"));
        Console.Write(" hex: {0}\n", Enum.Format(typeof(EmpType), e, "X"));
}
```

Результат работы этой программы (и предыдущих примеров, посвященных перечислениям) представлен на рис. 2.24.

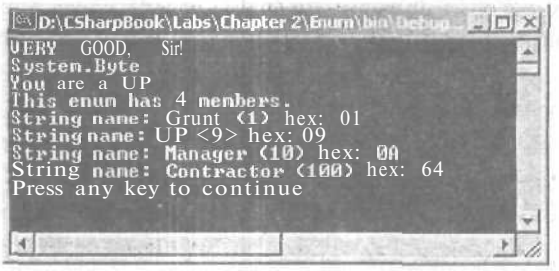


Рис. 2.24. Возможности System.Enum

Кроме того, в классе System.Enum предусмотрено очень полезное свойство IsDefined. Оно позволяет определить, является ли выбранная вами символьная строка элементом указанного перечисления. Например, предположим, что вам потребовалось узнать, является ли значение Salesperson элементом перечисления EmpType:

```
// Есть ли в EmpType элемент Salesperson?
if(Enum.IsDefined(typeof(EmpType),"SalesPerson"))
        Console.WriteLine("Yep, we have sales people.");
else
        Console.WriteLine("No, we have no profits...");
```

Последнее, о чем необходимо упомянуть в этом разделе: перечисления C# поддерживают работу с большим количеством перегруженных операторов, которые могут выполнять различные операции с числовыми значениями переменных. Например:

```
// Какому из этих двух переменных-членов перечисления соответствует
// большее числовое значение?
EmpType Joe = EmpType.VP;
EmpType Fran - EmpType.Grunt;

if(Joe < Fran)
        Console.WriteLine("Joe's value is less than Fran's");
else
        Console.WriteLine("Fran's value is less than Joe's");
```

Код приложения Enum можно найти в подкаталоге Chapter 2.

Определение структур в C#

Мы уже сталкивались со структурами в начале этой главы, однако эти конструкции C# заслуживают более подробного рассмотрения. Во многих отношениях структуры C# можно рассматривать как некую особую разновидность классов. С классами структуры роднит многое: для структур можно определять конструкторы (только принимающие параметры), структуры могут реализовывать интерфейсы, структуры могут содержать любое количество внутренних членов. Для структур C# не существует единого базового класса (тип System.Structure в C# не предусмотрен), однако косвенно все структуры являются производными от типа ValueType. Вспомним, что основное назначение ValueType заключается в том, чтобы обеспечить типы данных, производные от System.Object, членами для работы с этими типами данных как со структурными (value-based), когда значения передаются как локальные копии, а не как ссылки на области в памяти. Вот простой пример структуры C#:

```
// Вначале нам потребуется наше перечисление
enum EmpType ; byte
{
        Manager = 10. Grunt = 1. Contractor - 100. VP - 9
}

struct EMPLOYEE
{
```

```
        public EmpType title;        // Одно из полей структуры - перечисление,
                                     // определенное выше
        public string name;
        public short deptID;
}

class StructTester
{
        public static int Main(string[] args)
        {
                // Создаем и присваиваем значения Фреду
                EMPLOYEE fred;
                fred.deptID - 40:
                fred.name = "Fred";
                fred.title = EmpType.Grunt;
                return 0;
        }
}
```

Мы создали структуру EMPLOYEE (для нее была выделена память в области стека) и теперь можем обращаться к каждому члену структуры, используя формат имя_структуры.имя_члена. Вполне возможно, что в реальном приложении для более удобного присвоения значений членам структуры нам придется определить свой собственный конструктор или несколько конструкторов. В связи с этим необходимо помнить, что вы не можете переопределить конструктор для структуры по умолчанию — тот конструктор, который не принимает параметров. Все ваши конструкторы обязательно должны принимать один или несколько параметров:

```
// Для структур можно определить конструкторы, но все созданные вами конструкторы должны
принимать параметры
struct EMPLOYEE
{
        // Поля
        public EmpType title:
        public string name;
        public short deptID;

        // Конструктор
        public EMPLOYEE (EmpType et. string n. short d)
        {
                title - et;
                name = n;
                deptID - d;
        }
}
```

При помощи такого определения структуры, в котором предусмотрен конструктор, вы можете создавать новых сотрудников следующим образом:

```
class StructTester
{
        // Создаем Мэри и присваиваем ей значения при помощи конструктора
        public static int Main(string[] args)
        {
                // Для вызова нашего конструктора мы обязаны использовать
                // ключевое слово new
                EMPLOYEE mary = new EMPLOYEE(EmpType.VP, "Mary". 10):
                ...
```

```
        return 0;
    }
}
```

Конечно же, структуры могут быть использованы в качестве принимаемых и возвращаемых методами параметров. Например, пусть в нашем классе StructTester будет предусмотрен метод DisplayEmpStats():

```
// Извлекаем интересующую нас информацию из структуры EMPLOYEE
public void DisplayEmpStats(EMPLOYEE e)
{
    Console.WriteLine("Here is {0}\'s info:", e.name);
    Console.WriteLine("Department ID: {0}", e.deptID);
    Console.WriteLine("Title: {0}", Enum.Format(typeof(EmpType), e.title, "G"));
}
```

Так можно выполнить программу с использованием DisplayEmpStats():

```
// Выведен данные о Мэри и Фреде:
public static int Main(string[] args)
{
    ...
    StructTester t = new StructTester();
    t.DisplayEmpStats(mary);
    t.DisplayEmpStats(fred);

    return 0;
}
```

Результат работы этой программы представлен на рис. 2.25.

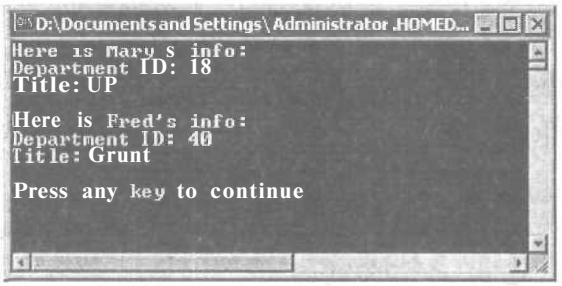


Рис. 2.25. Возможности структур C#

Еще раз об упаковке и распаковке

Как уже говорилось выше в этой главе, упаковка и распаковка — это наиболее удобный способ преобразования структурного типа в ссылочный, и наоборот. Основное назначение структур — возможность получения некоторых преимуществ объектной ориентации, но при более высокой производительности за счет размещения в стеке. Чтобы преобразовать структуру в ссылку на объект, необходимо упаковать ее экземпляр:

```
// Создаем и упаковываем нового сотрудника
EMPLOYEE Stan = new EMPLOYEE(EmpType.Grunt, "Stan", 10);
object stanInBox = stan;
```

Поскольку stanInBox относится к ссылочным типам данных, но при этом сохраняет внутренние значения исходного типа данных EMPLOYEE, вы можете использовать stan во всех случаях, когда вам нужен объект, и при необходимости производить распаковку:

```
// Поскольку мы ранее произвели упаковку данных, мы можем распаковать их
и производить операции с содержимым
public void UnboxThisEmployee(object o)
{
        // Производим распаковку в структуру EMPLOYEE для получения доступа
        // ко всем полям
        EMPLOYEE temp = (EMPLOYEE)o;
        Console.WriteLine(temp.name + "is alive!");
}
```

Вызов этого метода может выглядеть следующим образом:

```
// Передаем упакованного сотрудника на обработку
t.UnboxThisEmployee(stanInBox);
```

Вспомним, что компилятор C# при необходимости автоматически производит упаковку. Поэтому вполне допускается передать объект stan (относящийся к типу EMPLOYEE) методу UnboxThisEmployee() напрямую:

```
// Stan будет упакован автоматически
t.UnboxThisEmployee(stan);
```

Однако, поскольку вы определили метод UnboxThisEmployee() таким образом, что он обязан принимать объект в качестве параметра, вам придется распаковать эту ссылку для получения доступа к полям структуры EMPLOYEE.

Код приложения Structures можно найти в подкаталоге Chapter 2.

Определяем пользовательские пространства имен

К этому моменту мы уже создали множество небольших тестовых программ, которые работают с существующими пространствами имен во вселенной .NET. При построении реальных приложений часто бывает очень полезным сгруппировать ваши типы данных в специально созданные для этой цели пространства имен. В C# эта операция производится при помощи ключевого слова namespace.

Предположим, что мы создаем набор геометрических классов. Один класс получит название Square (квадрат), второй — Circle (окружность), а третий — Hexagon (шестиугольник). Поскольку все эти классы относятся к геометрическим фигурам, мы принимаем решение сгруппировать их в общем пользовательском пространстве имен. Основных вариантов в нашем распоряжении два. Согласно первому, мы можем определить все классы в одном файле (shapesLib.cs):

```
// shapeslib.cs
namespace MyShapes
{
        using System:

        // Класс Circle
        public class Circle { // Интересные методы }
```

```
        // Класс Hexagon
        public class Hexagon { // Более интересные методы }

        // Класс Square
        public class Square { // Еще более интересные методы }
}
```

Обратите внимание, что пространство имен MyShapes действует как контейнер для всех этих типов. Вы можете разбить единое пространство имен C# на несколько физических файлов. Для этого достаточно просто определить в разных файлах одно и то же пространство имен и поместить в него определения классов:

```
// circle.cs
namespace MyShapes
{
        using System;

        // Класс Circle
        class Circle { // Интересные методы }
}

// hexagon.cs
namespace MyShapes
{
        using System;

        // Класс Hexagon
        class Hexagon { // Более интересные методы }
}

// square.cs
namespace MyShapes
{
        using System:

        // Класс Square
        class Square { // Еще более интересные методы }
}
```

Если вам потребуется использовать эти классы внутри другого приложения, удобнее всего это сделать при помощи ключевого слова using:

```
// Используем объекты, определенные в другом пространстве имен
namespace MyApp
{
using System:
using MyShapes;

class ShapeTester
{
        public static void Main()
        {
                // Все эти объекты были определены в пространстве имен MyShapes
                Hexagon h = new Hexagon();
                Circle c = new Circle();
                Square s - new Square();
        }
}
}
```

Применение пространств имен для разрешения конфликтов между именами классов

Пространства имен могут быть использованы для разрешения конфликтов между именами объектов в тех ситуациях, когда в нашем приложении используются разные объекты с одинаковыми именами. Представим себе такую ситуацию: наш класс ShapeTester должен использовать новое пространство имен, которое называется My3DShapes. В этом пространстве имен определены три класса с теми же именами, но предназначенные для работы с трехмерными объектами:

```
// Еще одно пространство имен для геометрических фигур
namespace My3DShapes
{
        using System;

        // Класс 3D Circle
        class Circle{}

        // Класс 3D Hexagon
        class Hexagon{}

        // Класс 30 Square
        class Square{}
}
```

Если вы измените класс ShapeTester как показано ниже, результатом станут три ошибки компилятора, сообщающие о конфликте имен:

```
// В коде есть двусмысленности!
namespace MyApp
{
using System:
using MyShapes;
using My3DShapes;

class ShapeTester
        {
                public static void Main()
                {
                        // Неизвестно, к объектам какого пространства имен мы обращаемся
                        Hexagon h = new Hexagon();
                        Circle c = new Circle();
                        Square s = new Square();
                }
        }
}
```

Результатом, как мы уже говорили, станут сообщения компилятора об ошибках (рис. 2.26).

Проще всего избавиться от подобных конфликтов, указав для каждого класса его полное имя вместе с именем соответствующего пространства имен:

```
// Конфликтов больше нет
public static void MainO
{
        My3DShapes.Hexagon h = new My3DShapes.Hexagon();
        My3DShapes.Circle c = new My3DShapes.Circle();
        MyShapes.Square s - new MyShapes.Square();
}
```

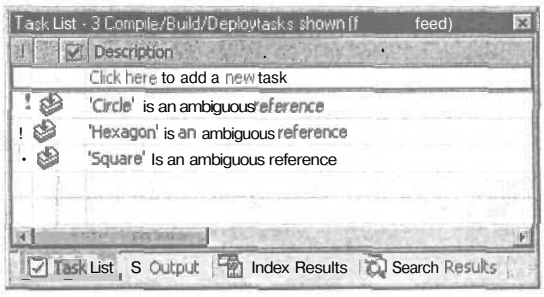


Рис. 2.26. Конфликты имен

Использование псевдонимов для имен классов

Еще одна возможность избавиться от конфликтов имен — использовать для имен классов псевдонимы. Например:

```
namespace MyApp
{
using System;
using MyShapes;
using My3DShapes;

// Создаем псевдоним для класса из другого пространства имен
using The3DHexagon = My3DShapes.Hexagon;

class ShapeTester
    {
        public static void Main()
        {
            Hexagon h = new Hexagon();
            Circle c = new Circle();
            Square s = new Square();

            // Создаем объект при помощи псевдонима
            The3DHexagon h2 = new The3DHexagon();
        }
    }
}
```

Вложенные пространства имен

Последнее, что необходимо сообщить о пространствах имен, — то, что вы можете без каких-либо ограничений вкладывать одни пространства имен в другие. Такой подход очень часто используется в библиотеках базовых классов .NET для организации типов. Например, если вы хотите создать пространство имен более высокого уровня, в котором будет находиться наше пространство имен My3DShapes, это можно сделать следующим образом:

```
// Классы для геометрических фигур расположены в пространстве
имен Chapter2Types.My3DShapes
namespace Chapter2Types
I
```

```
namespace My3DShapes
{
        using System;

        // класс 3D Circle
        class Circle{}

        // Класс 3D Hexagon
        class Hexagon{}

        // Класс 30 Square
        class Square{}
}
}
```

Код приложения Namespaces можно найти в подкаталоге Chapter 2.

Подведение итогов

В этой главе были рассмотрены основы языка программирования C# — те его составные части, которые используются практически в каждом приложении, вне зависимости от его конкретного типа.

В каждой программе на языке C# должен быть определен класс со статическим методом Main(). Этот метод используется как точка входа для приложения. Внутри метода MainC) обычно производится создание объектов и различные операции с ними — то, что составляет жизнь любого приложения.

Каждому встроенному типу данных C# соответствует свой тип в пространстве имен System. Каждый встроенный тип данных наследует большое количество методов, которые во многом упрощают работу с данными этого типа.

В этой главе мы также познакомились с такими конструкциями языка C#, как массивы, строковые значения и перечисления и рассмотрели наиболее важные их возможности. В C# существуют средства для упаковки (превращения структурного типа данных в ссылочный) и распаковки (проведения обратной операции). В конце главы были рассмотрены вопросы, связанные с созданием и применением пользовательских пространств имен.

C# и объектно-ориентированное программирование

3

В предыдущих главах мы познакомились с основополагающими принципами и конструкциями C#. В этой главе мы погрузимся в изучение подробностей объектно-ориентированного программирования (ООП) с применением средств C#. Мы начнем с рассмотрения знаменитых «столпов ООП» — инкапсуляции, наследования и полиморфизма, включая особенности их воплощения в C#. В результате мы должны получить знания, достаточные для построения иерархий классов C#.

Кроме того, мы познакомимся с такими возможностями, как указание области видимости на уровне типа (а не члена типа), создание пользовательских свойств и разработка «закрытых» классов, Вы также получите представление об использовании структурной обработки исключений для борьбы с ошибками времени выполнения. В конце главы будет рассмотрен механизм работы «управляемой кучи», включая программные средства взаимодействия со сборщиком мусора .NET с использованием статических методов, определенных в System.GC.

Формальное определение класса в C#

Класс в C#, как и в других языках программирования, — это пользовательский тип данных (user defined type, UDT), который состоит из данных (часто называемых атрибутами или свойствами) и функциями для выполнения с этими данными различных действий (эти функции обычно называются методами). Классы позволяют группировать в единое целое данные и функциональность, моделируя объекты реального мира. Именно это свойство классов и обеспечивает одно из наиболее важных преимуществ объектно-ориентированных языков программирования.

К примеру, предположим, что нам потребовалось создать модель сотрудника нашей организации. Конечно же, для этой цели удобнее всего создать специальный класс. Этот класс, как минимум, должен хранить данные об имени работника, его идентификационном номере и текущей заработной плате. Помимо этого, пусть в нашем классе будут определены два метода — `GiveBonus()` для увеличения зара-

ботной платы сотрудника и DisplayStats() для вывода всех имеющихся данных об этом сотруднике (рис. 3.1).

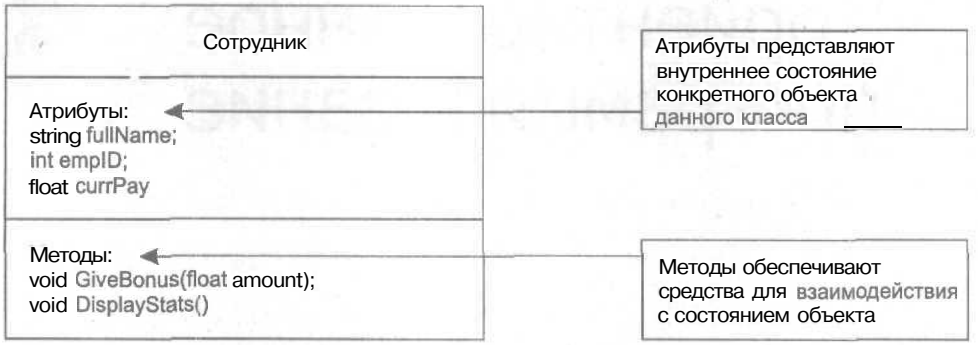


Рис. 3.1. Определение простого класса

Как уже говорилось в главе 2, для классов C# можно определить любое количество конструкторов. Эти специальные методы классов позволяют создавать объекты данного класса, настраивая при этом их исходное состояние нужным вам образом. Любой класс C# автоматически снабжается конструктором по умолчанию (не принимающим никаких параметров). Этот конструктор в C# (в отличие от C++) при создании объекта автоматически присвоит всем переменным-членам класса безопасные значения по умолчанию. Кроме того, вы можете как переопределить конструктор по умолчанию, так и создать такое количество пользовательских конструкторов (принимающих параметры), сколько вам необходимо. Давайте определим наш класс для сотрудника на C#:

```
// Исходное определение класса
class Employee
{
    // Внутренние закрытые данные класса
    private string fullName;
    private int empID;
    private float currPay;

    // Конструкторы
    public Employee() {}
    public Employee(string fullName, int empID, float currPay)
    {
        this.fullName = fullName;
        this.empID = empID;
        this.currPay = currPay;
    }

    // Метод для увеличения зарплаты сотрудника
    public void GiveBonus(float amount)
    { currPay += amount; }

    // Метод для вывода сведений о текущем состоянии объекта
    public virtual void DisplayStats()
    {
```

```
            Console.WriteLine("Name: {0}", fullName);
            Console.WriteLine("Pay: {0}", currPay);
            Console.WriteLine("ID: {0}", empID);
            Console.WriteLine("SSN: {0}", ssn);
        }
}
```

Обратите внимание на то, что определение для конструктора по умолчанию не содержит каких-либо действий, которые должен выполнять конструктор:

```
class Employee
{
        // Всем переменным-членам значения по умолчанию будут присвоены автоматически
        public Employee() {}
...
}
```

Необходимо обязательно помнить о следующем обстоятельстве: в C# (как и в C++), если вы определили хотя бы один пользовательский конструктор (принимающий параметры), конструктор по умолчанию автоматически создаваться уже не будет и его придется определять явно. В противном случае при попытке выполнения такого, к примеру, кода:

```
// Вызываем конструктор по умолчанию
Employee e = new Employee();
```

вы получите сообщение об ошибке компилятора.

Использование пользовательского конструктора очевидно:

```
// Вызываем пользовательский конструктор (двумя способами)
public static void Main()
{
        Employee e = new Employee("Joe", 80, 30000);
        e.GiveBonus(200);

        Employee e2;
        e2 = new Employee("Beth", 81, 50000);
        e2.GiveBonus(1000);
        e2.DisplayStats();
}
```

Код приложения Employees (с которым мы будем работать в продолжение всей этой главы) можно найти в подкаталоге Chapter 3.

Ссылки на самого себя

В определениях пользовательских конструкторов часто используется зарезервированное слово this:

```
// В C#, как и в C++, и в Java, предусмотрено использование зарезервированного
// слова this
public Employee(string fullName, int empID, float currPay)
{
        this.fullName = fullName;
        this.empID = empID;
        this.currPay = currPay;
}
```

В C# слово this используется для ссылки на текущий экземпляр объекта. В Visual Basic схожие функции выполняет зарезервированное слово Me, а про-

граммисты, использующиеC++ и Java, вообще должны почувствовать себя как дома, поскольку в этих языках также есть слово this, выполняющее те же функции.

Главная причина, по которой в конструкторах используется слово this, — желание избежать конфликтов между именами принимаемых параметров и именами внутренних переменных-членов класса. Конечно же, такого конфликта можно избежать и более простым способом — определить для принимаемых переменных имена, отличные от имен переменных-членов класса. Кроме того, необходимо помнить, что через указатель this невозможно обратиться к статическим функциям-членам. Причина очевидна: такие функции принадлежат не отдельным объектам класса, а всему классу в целом.

Перенаправление вызовов конструктора с использованием слова this

Зарезервированное слово this в C# может быть использовано еще для одной цели; для передачи вызова от одного конструктора к другому. Вот пример:

```
class Employee
{
        public Employee(string fullName, int empID, float currPay)
        {
                this.fullName - fullname:
                this.empID = empID;
                this.currPay - currPay:
        }
        // Если пользователь вызовет этот конструктор, перенаправить вызов варианту
        // с тремя параметрами
        public Employee(string full Name)
                :this(fullName, IDGenerator.GetNewEmpID(), 0.0F) {}
    ...
}
```

В нашем классе Employee теперь определено два пользовательских конструктора: первый принимает три параметра, а второй требует единственного параметра (имени сотрудника). Однако для полного создания объекта Employee вам необходимо убедиться, что для объекта будет надлежащим образом создан идентификатор сотрудника (empID) и установлена ставка оплаты. Предположим, что в нашем распоряжении имеется еще один класс, который называется IDGenerator. В этом классе определен статический метод GetNewEmpID, который используется для создания нового идентификатора сотрудника. С помощью этого метода, а также подставив недостающий третий параметр, мы переадресуем вызов первому конструктору — тому, который принимает три параметра. Без такого перенаправления нам пришлось бы вписывать во второй конструктор повторяющийся код:

```
// currPay будет автоматически присвоено значение 0.0F - поскольку это - значение
// по умолчанию для этого типа данных
public Employee(string fullName)
{
        this.fullName - fullName;
        this.empID - IDGenerator.GetNewEmpID();
}
```

Определение открытого интерфейса по умолчанию

Открытый интерфейс по умолчанию для класса (default public interface) — это набор открытых членов данного класса. С точки зрения обращения к членам класса из остальной части программы открытый интерфейс по умолчанию — это набор тех переменных, свойств и методов — членов, к которым можно обратиться в формате имя_объекта.член_класса. С точки зрения определения класса — это набор тех членов класса, которые объявлены с использованием ключевого слова public. В C# открытый интерфейс по умолчанию для класса может состоять из следующих членов этого класса:

- методов — наборов действий, определенных как единое целое, с помощью которых реализуется поведение данного класса;
- свойств — «переодетых» функций для получения данных и их изменения;
- полей: открытых переменных-членов (как правило, использование таких переменных — не самый лучший вариант, но в C# они поддерживаются).

Как мы увидим в главе 5, открытый интерфейс класса по умолчанию может также быть настроен таким образом, чтобы поддерживать события. Однако сейчас мы оставим события в стороне и сконцентрируемся на использовании свойств, методов и полей.

Указание области видимости на уровне типа: открытые и внутренние типы

Перед тем как углубиться в новые варианты нашего примера с сотрудником, нам необходимо разобраться с областями видимости для типов. В предыдущих главах мы много раз встречались с определениями классов вида;

```
class HelloClass
{
        // Любое количество методов с любым количеством параметров
        // Конструктор по умолчанию и пользовательские конструкторы
        // Если в этом классе определена и точка входа для всей программы,
        // то еще и статический метод Main()
}
```

Как мы помним, для каждого члена класса необходимо указать (или принять значение по умолчанию) область видимости с помощью одного из следующих ключевых слов: public, private, protected и internal. Однако область видимости задается не только для членов класса, но и для самих классов! Разница между областью видимости для членов классов и областью видимости для самих классов заключается в том, что в первом случае мы определяем те члены класса, которые будут доступны через экземпляры объектов, а во втором — те области программы, из которых будет возможно обращение к данным классам.

Для класса в C# используется только два ключевых слова для определения области видимости: public и internal. Объекты классов, определенных как public (открытых), могут быть созданы как из своего собственного двоичного файла, так и из других двоичных файлов C# (то есть из другой сборки). Если переопределять HelloClass как public, то это можно сделать следующим образом:

```
// Теперь этот класс можно создавать из-за пределов сборки, в которой он определен
public class HelloClass
{
        // Любое количество методов с любым количеством параметров
        // Конструктор по умолчанию и пользовательские конструкторы
        // Если в этом классе определена и точка входа для всей программы,
        // то еще и статический метод Main()
}
```

Если вы не указали явно область видимости для класса, то по умолчанию этот класс будет объявлен как Internal (внутренний). Объекты таких классов могут создаваться только объектами из той же сборки, в которой они были определены. Внутренние классы часто рассматриваются как вспомогательные (helper classes), поскольку они используются типами данной сборки для помощи в выполнении каких-либо действий.

```
// Внутренние классы могут использоваться только внутри той сборки, а которой они
определены
internal class HelloClassHelper
{
        ...
}
```

Классы — это не единственные пользовательские типы данных, для которых существует атрибут области видимости. Как мы помним, тип — это общий термин, который используется в отношении классов, структур, перечислений, интерфейсов и делегатов. Атрибут видимости (public или internal) может быть установлен для любого типа C#, например:

```
namespace HelloClass
{
using System;
internal struct X                // Эта структура не сможет быть использована из-за пределов
                                 // данной сборки
{
        private int myX;
        public int GetMyX() {return MyX; }
        public X(int x) { myX = x;}
}

internal enum Letters            // Это перечисление не сможет быть использовано
                                 // из-за пределов данной сборки
{
        a = 0, b = 1, c = 2
}

public class HelloClass          // Можно использовать откуда угодно
{
        public static int Main(string[] args)
        {
                X theX - new X(26);
                Console.WriteLine(theX.GetMyXO + "\n" + Letters.b.ToString());
                return 0;
        }
}
}
```

Все рассмотренные выше типы можно для наглядности собрать в единую схему (рис. 3.2):

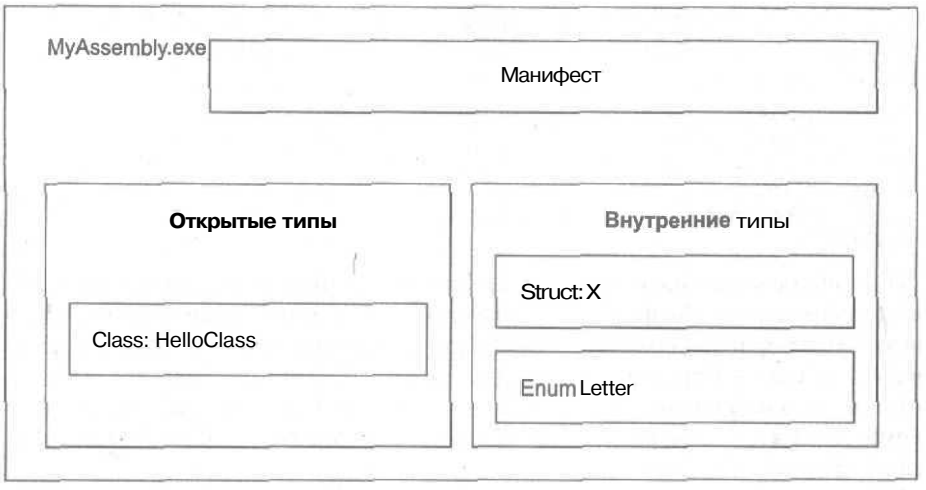


Рис. 3.2. Внутренние и открытые типы

Вопросы, связанные с созданием сборок .NET, рассматриваются в главе 6. Сейчас же нам достаточно помнить, что все типы данных в C# могут объявляться как открытые (public) — доступные из-за пределов собственной сборки, и как закрытые (private) — доступные только из данной сборки.

Столпы объектно-ориентированного программирования

C# можно считать новым членом сообщества объектно-ориентированных языков программирования, к самым распространенным из которых относятся Java, C++, Object Pascal и (с некоторыми допущениями) Visual Basic 6.0. В любом объектно-ориентированном языке программирования обязательно реализованы три важнейших принципа — «столпа» объектно-ориентированного программирования:

- инкапсуляция: как объекты прячут свое внутреннее устройство;
- наследование: как в этом языке поддерживается повторное использование кода;
- полиморфизм: как в этом языке реализована поддержка выполнения нужного действия в зависимости от типа передаваемого объекта?

Конечно же, все три этих принципа реализованы и в C#. Однако перед тем как начать рассматривать синтаксические особенности реализации этих принципов, мы постараемся убедиться, что в них не осталось абсолютно никаких неясностей.

Инкапсуляция

Первый «столп» объектно-ориентированного программирования — это инкапсуляция. Так называется способность прятать детали реализации объектов от пользо-

вателей этих объектов. Например, предположим, что вы создали класс с именем DBReader (для работы с базой данных), в котором определено два главных метода: Open() и Close().

```
// Класс DBReader скрывает за счет инкапсуляции подробности открытия
// и закрытия баз данных
DBReader f - new DBReader();
f.Open(@"C:\foo.mdf");
      // Выполняем с базой данных нужные нам действия
f.Close();
```

Наш вымышленный класс DBReader инкапсулирует внутренние подробности того, как именно он обнаруживает, загружает, выполняет операции и в конце концов закрывает файл данных. За счет инкапсуляции программирование становится проще: вам нет необходимости беспокоиться об огромном количестве строк кода, которые выполняют свою задачу скрыто от вас. Все, что от вас требуется — создать экземпляр нужного класса и передать ему необходимые сообщения (типа «открыть файл с именем foo.mdf»).

С философией инкапсуляции тесно связан еще один принцип — сокрытия всех внутренних данных (то есть переменных-членов) класса. Как мы уже говорили, в идеале все внутренние переменные члены класса должны быть определены как private. В результате обращение к ним из внешнего мира будет возможно только при помощи открытых функций — членов. Такое решение, помимо всего прочего, защищает вас от возможных ошибок, поскольку открытые данные очень просто повредить.

Наследование: отношения «быть» и «иметь»

Следующий столп объектно-ориентированного программирования — наследование. Под ним понимается возможность создавать новые определения классов на основе существующих. В сущности, наследование позволяет вам расширять возможности, унаследованные от базового класса, в собственном производном классе. Простой пример реализации такого подхода представлен на рис. 3.3.

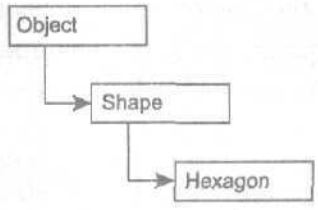


Рис. 3.3. Отношение «быть»

Как мы помним, на вершине любой иерархии в .NET всегда находится базовый класс Object. В нашей ситуации возможности этого класса вначале дополняются возможностями, привнесенными классом Shape. Речь идет о свойствах, полях, методах и событиях, которые являются общими для всех геометрических фигур (shapes). Класс Hexagon (шестиугольник), производный от Shape, дополняет возможности предыдущих двух базовых классов своими собственными уникальными свойствами.

Диаграмму, представленную на рис. 3.3, можно прочесть следующим образом: «Шестиугольник есть геометрическая фигура, которая есть объект». Когда ваши классы связываются друг с другом отношениями наследования, это означает, что вы устанавливаете между ними отношения типа «быть» (is-a). Такой тип отношений называется также классическим наследованием.

В мире объектно-ориентированного программирования используется еще одна форма повторного использования кода. Эта форма называется включением–делегированием (или отношением «иметь» —has-a). При ее использовании один класс включает в свой состав другой и открывает внешнему миру часть возможностей этого внутреннего класса.

Например, если вы создаете программную модель автомобиля, у вас может появиться идея включить внутрь объекта «автомобиль» объект «радио» с помощью отношения «иметь». Это вполне разумный подход, поскольку вряд ли возможно произвести как радио от автомобиля, так и автомобиль от радио, используя отношения наследования. Вместо наследования вы создаете два независимых класса, работающих совместно, где внешний (контейнерный) класс создает внутренний класс и открывает внешнему миру его возможности (рис. 3.4).

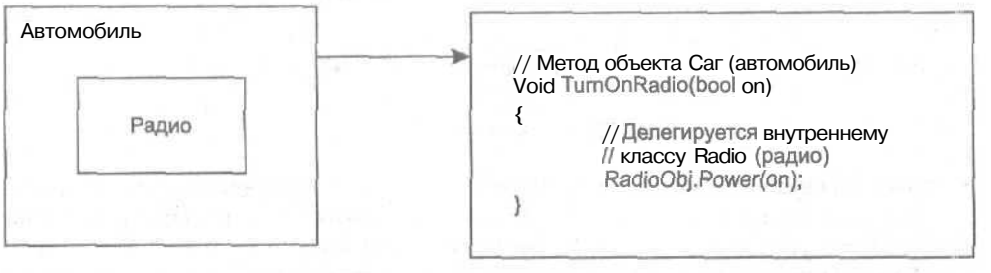


Рис. 3.4. Отношения между средой выполнения и библиотекой базовых классов .NET

Внешний объект Car (автомобиль) ответственен за создание внутреннего объекта Radio (радио). Если вы хотите, чтобы через объект Car можно было бы обращаться ко внутреннему объекту Radio, в классе Car необходимо предусмотреть специальный открытый интерфейс для этой цели. Обратите внимание, что пользователь объекта Car и не узнает, что тот передает обращения какому-то внутреннему объекту:

```
// Внутренний класс Radio инкапсулирован внешним классов Car
Car viper - new Car();
viper.TurnOnRadio(false);    // Вызов будет передан внутреннему объекту Radio
```

Полиморфизм: классический и для конкретного случая

Последний, третий столп объектно-ориентированного программирования — это полиморфизм. Можно сказать, что этот термин определяет возможности, заложенные в языке, по интерпретации связанных объектов одинаковым образом. Существует две основные разновидности полиморфизма: классический полиморфизм и полиморфизм «для конкретного случая» (ad hoc). Классический полиморфизм встречается только в тех языках, которые поддерживают классическое наследование (в том числе, конечно, и в C#), При классическом полиморфизме вы можете

определить в базовом классе набор членов, которые могут быть замещены в производном классе. При замещении в производных классах членов базового класса эти производные классы будут по-разному реагировать на одни и те же обращения.

Для примера мы вновь обратимся к нашей иерархии геометрических фигур. Предположим, что в классе Shape (геометрическая фигура) определена функция `Draw()` — рисование, которая не принимает параметров и ничего не возвращает. Поскольку геометрические фигуры бывают разными и каждый тип фигуры потребуется изображать своим собственным уникальным способом, скорее всего, нам потребуется в производных классах (таких как Hexagon — шестиугольник и `Circle` — окружность) создать свой собственный метод `Draw()`, заместив им метод `Draw()` базового класса (рис. 3.5).

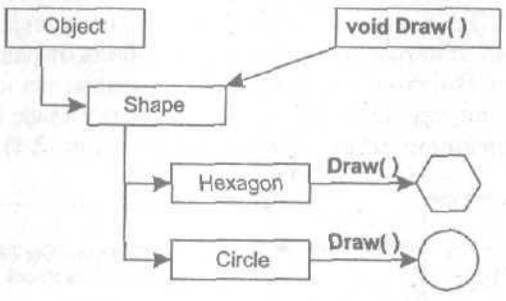


Рис. 3.5. Классический полиморфизм

Классический полиморфизм позволяет определять возможности всех производных классов при создании базового класса. Например, в нашем примере вы можете быть уверены, что метод `Draw()` в том или ином варианте присутствует в любом классе, производном от Shape. К достоинствам классического полиморфизма можно отнести также и то, что во многих ситуациях вы сможете избежать создания повторяющихся методов для выполнения схожих операций (типа `DrawCircle()`, `DrawRectangle()`, `DrawHexagon()` и т. д.).

Вторая разновидность полиморфизма — *полиморфизм для конкретного случая* (ad hoc polymorphism). Этот тип полиморфизма позволяет обращаться схожим образом к объектам, *не связанным классическим наследованием*. Достигается это очень просто: в каждом из таких объектов должен быть метод с одинаковой сигнатурой (то есть одинаковым именем метода, принимаемыми параметрами и типом возвращаемого значения. В языках, поддерживающих полиморфизм этого типа, применяется технология «позднего связывания» (late binding), когда тип объекта, к которому происходит обращение, становится ясен только в процессе выполнения программы. В зависимости от того, к какому типу мы обращаемся, вызывается нужный метод. В качестве примера рассмотрим схему на рис. 3.6.

Обратите внимание, что общего предка — базового класса для `CCircle`, CHexagon и CRectangle не существует. Однако в каждом классе предусмотрен метод `Draw()` с одинаковой сигнатурой. Для того чтобы продемонстрировать применение полиморфизма этого типа в реальном коде, мы воспользуемся примером на Visual Basic 6.0. До изобретения VB.NET Visual Basic не поддерживал классический полиморфизм (так же, как и классическое наследование), заставляя разработчиков сосредоточивать свои усилия на полиморфизме ad hoc.

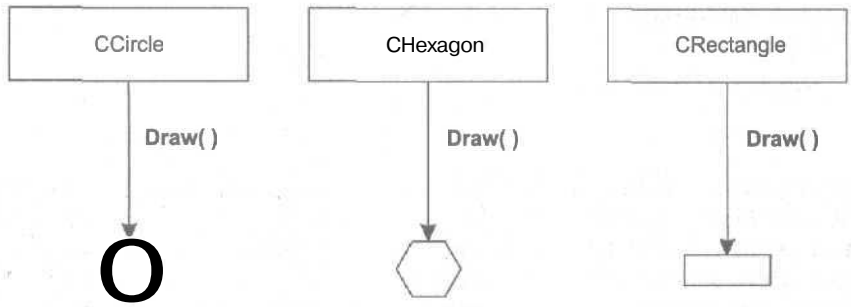


Рис. 3.6. Полиморфизм для конкретного случая

```
' Это - код на Visual Basic 6.0!
' Вначале создадим нассив элементов типа Object и установим для каждого элемента
ссылку на объект
Dim objArr(3) as Object
Set objArr(0) = New CCircle
Set objArr(1) = New CHexagon
Set objArr(2) = New CCircle
Set objArr(3) = New CRectangle

' Теперь с помощью цикла заставим каждый элемент нарисовать самого себя
Dim i as Integer
For i = 0 to 3
       objArr(i).Draw 'Позднее связывание
Next 1
```

В этом коде мы вначале создали массив элементов типа Object (это встроенный тип данных Visual Basic 6.0 для хранения ссылок на любые объекты, не имеющий ничего общего с классом System.Object в .NET). Затем мы связали каждый элемент массива с объектом соответствующего типа, а потом при помощи цикла воспользовались методом Draw() для каждого из этих объектов. Обратите внимание, что у геометрических фигур — элементов массива — нет общего базового класса с реализацией метода Draw() по умолчанию.

Теоретический обзор главных принципов полиморфизма — инкапсуляции, наследования и полиморфизма на этом закончен. Конечно же, в C# реализованы все эти принципы, при этом C# поддерживает и отношения «быть» и отношения «иметь» для повторного использования кода, и обе разновидности полиморфизма. Теперь наша задача — узнать, как реализуется каждый из этих принципов средствами синтаксиса C#.

Средства инкапсуляции в C#

Принцип инкапсуляции предполагает, что ко внутренним данным объекта (переменным-членам) нельзя обратиться напрямую через экземпляр этого объекта. Вместо этого для получения информации о внутреннем состоянии объекта и внесения изменений необходимо использовать специальные методы. В C# инкапсуляция реализуется на уровне синтаксиса при помощи ключевых слов public, private и protected. В качестве примера мы рассмотрим следующее определение класса:

```
// Класс с единственным полем
public class Book
{
      public int numberOfPages;
```

Г

Термин «поле» (field) используется для открытых данных класса — переменных, объявленных с ключевым словом public. При использовании полей в приложении возникает проблема: полю можно присвоить любое значение, а организовать проверку этого значения бизнес-логике вашего приложения достаточно сложно. Например, для нашей открытой переменной numberOfPages используется тип данных int. Максимальное значение для этого типа данных — это достаточно большое число (2 147 483 647). Если в программе будет существовать такой код, проблем со стороны компилятора не возникнет:

```
// Задумаемся...
public static void Main()
{
      Book miniNovel = new Book();
      miniNovel.numberOfPages = 30000000;
}
```

Тип данных int вполне позволяет указать для книги небольших размеров количество страниц, равное 30 000 000. Однако понятно, что книг такой величины не бывает, и во избежание дальнейших проблем желательно использовать какой-нибудь механизм проверки, который отсеивал бы явно нереальные значения (например, он пропускал бы только значения между 1 и 2000). Применение поля — открытой переменной не дает нам возможности простым способом реализовать подобный механизм. Поэтому поля в реальных рабочих приложениях используются нечасто.

Следование принципу инкапсуляции позволяет защитить внутренние данные класса от неумышленного повреждения. Для этого достаточно все внутренние данные сделать закрытыми (объявив внутренние переменные с использованием ключевых слов private или protected). Для обращения к внутренним данным можно использовать один из двух способов:

- создать традиционную пару методов — один для получения информации (accessor), второй — для внесения изменений (mutator);
- определить именованное свойство.

Еще один метод защиты данных, предлагаемый C#, — использовать ключевое слово readonly. Однако какой бы способ вы ни выбрали, общий принцип остается тем же самым — инкапсулированный класс должен прятать детали своей реализации от внешнего мира. Такой подход часто называется «программированием по методу черного ящика». Еще одно преимущество такого подхода заключается в том, что вы можете как угодно совершенствовать внутреннюю реализацию своего класса, полностью изменяя его содержимое. Единственное, о чем вам придется позаботиться, — чтобы в новой реализации остались методы с той же сигнатурой и функциональностью, что и в предыдущих версиях. В этом случае вам не придется менять ни строчки существующего кода за пределами данного класса.

Реализация инкапсуляции при помощи традиционных методов доступа и изменения

Давайте вновь обратимся к нашему классу Employee. Если мы хотим, чтобы внешний мир смог работать с внутренними данными этого класса (пусть это будет только одна переменная fullName, для которой используется тип данных string), то традиционный подход рекомендует создание метода доступа (accessor, или get method) и метода изменения (mutator, или set method). Набор таких методов может выглядеть следующим образом:

```
// Определение традиционных методов доступа и изменения для закрытой переменной
public class Employee
{
        private string fullName;
...
        // Метод доступа
        public string GetFullName() {return fullName; }

        // Метод изменения
        public void SetFullName (string n).
        {
                // Логика для удаления неположенных символов (!, @, #, $. % и прочих
                // Логика для проверки максимальной длины и прочего
                fullName = n;
        }
}
```

Такой подход требует наличия двух методов для взаимодействия с каждой из переменных. Вызов этих методов может выглядеть следующим образом:

```
// Применение методов доступа и изменения
public static int Main(string[] args)
{
        Employee p = new Employee();
        p.SetFullName("Fred");
        Console.WriteLine("Employee is named: " + p.GetFullName());

        // Ошибка! К закрытым данным нельзя обращаться напрямую
        // через экземпляр объекта!
        // p.FullName;
        return 0;
}
```

Второй способ инкапсуляции: применение свойств класса

Помимо традиционных методов доступа и изменения для обращения к закрытым членам класса можно также использовать свойства (properties). В Visual Basic и COM свойства — это привычный инструмент. Свойства позволяют имитировать доступ к внутренним данным класса: при получении информации или внесении изменений через свойство синтаксис выглядит так же, как при обращении к обычной открытой переменной. Но на самом деле любое свойство состоит из двух скрытых внутренних методов. Преимущество свойств заключается в том, что вместо того, чтобы использовать два отдельных метода, пользователь класса может использовать единственное свойство, работая с ним так же, как и с открытой переменной-членом данного класса:

```
// Обращение к имени сотрудника через свойство
public static int Main(string[] args)
{
        Employee p = new Employee();

        // Устанавливаем значение
        p.EmpID = 81;

        // Получаем значение
        Console.WriteLine("Person ID is: {0}", p.EmpID);
        return 0;
}
```

Если заглянуть внутрь определения класса, то свойства всегда отображаются в «реальные» методы доступа и изменения. А уже в определении этих методов вы можете реализовать любую логику (например, для устранения лишних символов, проверки допустимости вводимых числовых значений и прочего). Ниже представлен синтаксис класса Employee с определением свойства EmpID:

```
// Пользовательское свойство EmpID для доступа к переменной empID
public class Employee
{
...
        private int етрШ:

        // Свойство для empID
        public int EmpID
        {
                get {return empID;}
                set
                {
                        // Здесь вы можете реализовать логику для проверки вводимых
                        // значений и выполнения других действий
                        empID = value;
                } .
        }
}
```

Свойство C# состоит из двух блоков — блока доступа (get block) и блока изменения (set block). Ключевое слово value представляет правую часть выражения при присвоении значения посредством свойства. Как и все в C#, то, что представлено словом value — это также объект. Совместимость того, что передается свойству как value, с самим свойством, зависит от определения свойства. Например, свойство EmpID предназначено (согласно своему определению в классе) для работы с закрытым целочисленным empID, поэтому число 81 вполне его устроит:

```
// В данной случае типом данных, используемым для value, будет int
e3.EmpID - 81:
```

Показать дополнительные возможности применения ключевого слова value можно на таком примере:

```
// Свойство для empID
public int EmpID
{
        get {return empID;}
        set
        {
```

```
            // Как еще можно использовать value
            Console.WriteLine("value is the Instance of: {0}", value.GetType());

            Console.WriteLine("value as string: {0}", value.ToString());

            empID = value;
        }
    }
```

Результат работы данной программы представлен на рис. 3.7.

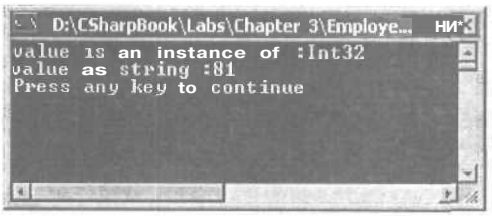


Рис. 3.7. Значение «value» при EmpID = 81

Необходимо отметить, что обращаться к объекту value можно только в пределах программного блока set внутри определения свойства. Попытка обратиться к этому объекту из любого другого места приведет к ошибке компилятора.

Последнее, что мы отметим — использование свойств (по сравнению с традиционными методами доступа и изменения) делает применение ваших типов более простым. Например, предположим, что в классе Employee предусмотрена внутренняя закрытая переменная для хранения информации о возрасте сотрудника. Вы хотите, чтобы при наступлении дня рождения этого сотрудника значение этой переменной увеличивалось на единицу. При использовании традиционных методов доступа и изменения эта операция будет выглядеть так:

```
Employee joe = new Employee();
joe.SetAge( joe.GetAge() + 1);
```

Используя свойство, вы можете сделать это проще:

```
Employee joe = new EmployeeO;
joe.Age++;
```

Внутреннее представление свойств C#

Многие программисты стараются использовать для имен методов доступа и изменения соответственно приставки get_ и set_ (например, get_Name() и set_Name()). Само по себе это не представляет проблемы. Проблему представляет другое: C# для внутреннего представления свойства использует методы с теми же самыми префиксами. К примеру, если вы откроете сборку Employees.exe при помощи утилиты ILDastn.exe, вы увидите, что каждому свойству соответствуют два отдельных (и скрытых) метода (рис. 3.8).

Поэтому подобное определение класса вызовет ошибку компилятора:

```
// Помните, что свойство C# автоматически отображается в пару методов get/set
public class Employee
{
...
```

```
        // Определение свойства
        public string SSN
        {
                get { return ssn; }  // Отображается в get_SSN()
                set { ssn = value; } // Отображается в set_SSN()
        }
        // Ошибка! Эти методы уже определены через свойство SSN!
        public string get_SSN() { return ssn; }
        public string set_SSN(string val) { ssn = val; }
}
```

Свойство автоматически отображается в два метода, но обратное утверждение не будет верным. Если вы создадите два метода get_X() и set_X(), свойство X автоматически создано не будет:

```
// Считаем, что в классе Foo определены два метода - get_X() и set_X(),
// а свойство X не определено
Foo f = new Foo();
f.X = 100;                    // Ошибка! X необходимо явно определить как свойство C#
Console.WriteLine(f.X);       // Ошибка - по той же самой причине
```

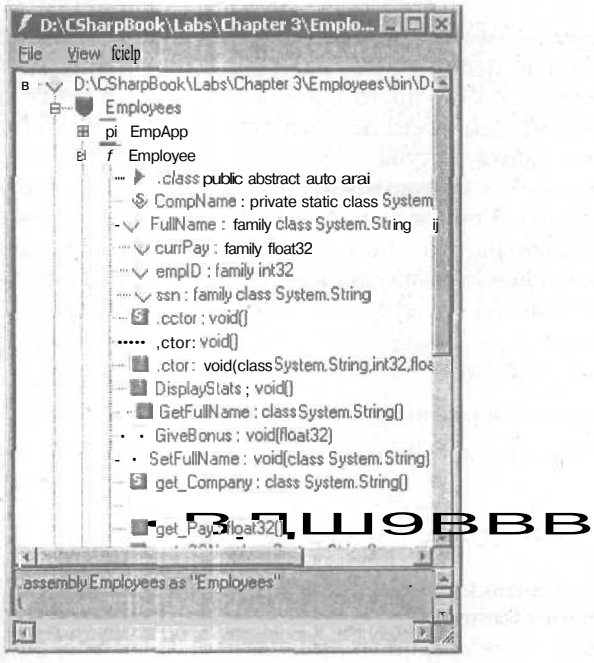


Рис. 3.8. Свойства отображаются в скрытые методы get_ и set_

Свойства только для чтения, только для записи и статические

Наш рассказ о свойствах классов C# будет неполон, если мы не упомянем еще о некоторых связанных с ними моментах. Как мы помним, наше свойство EmpID было создано как свойство, доступное и для чтения, и для записи. Однако при со-

здании пользовательских свойств класса часто возникает необходимость создать свойство, которое будет доступно только для чтения. Делается это очень просто: необходимо в определении свойства пропустить блок set. В результате свойство станет доступным только для чтения:

```
public class Employee
{
...
        // Будем считать, что исходное значение этой переменной присваивается с помощью
        // конструктора класса
        private string ssn;

        // А вот так выглядит свойство только для чтения
        public string SSN ( get { return ssn: } }
        ...
}
```

C# также поддерживает статические свойства. Как мы помним, статические переменные предназначены для хранения информации на уровне всего класса, а не его отдельных объектов. Если у нас объявлены статические данные (то есть те же переменные), то обращаться к ним и устанавливать значения должны статические свойства. Предположим, что в нашем классе Employee мы собираемся, помимо всего прочего, хранить еще и информацию об имени организации, в которой работают все сотрудники — объекты класса Employee. Для этого будет использована специальная статическая переменная. Соответствующее статическое свойство для работы с этой переменной может выглядеть так:

```
// Со статическими данными должны работать статические свойства
public class Employee
{
        private static string CompName;     // Статическая переменная
        public static string Company        // Статическое свойство
        {
                get { return CompName; }
                set { CompName = value: }
        }
        ...
}
```

Обращение к статическим свойствам производится так же, как и к статическим методам;

```
// Задаем и получаем информацию об имени компании
public static int Main(string[] args)
{
        Employee.Company = "Intertech, Inc";
        Console.WriteLine("These folks work at {0}", Employee.Company);
        ...
}
```

Статические конструкторы

Само словосочетание «статический конструктор» звучит несколько странно — ведь мы знаем, что конструкторы нужны для создания объектов, а все, что имеет определение «статический», работает на уровне классов, а не отдельных объектов. Однако в C# такие конструкторы вполне имеют право на существование. Единственное их назначение — присваивать исходные значения статическим переменным.

С точки зрения синтаксиса статические конструкторы — это достаточно причудливые образования. Например, для них нельзя использовать модификаторы области видимости, однако слово static должно присутствовать обязательно. Вот пример ситуации, в которой нам может пригодиться статический конструктор: предположим, что мы желаем, чтобы статической переменной CompName всегда при создании присваивалось значение InterTech, Inc. Мы можем сделать это следующим образом:

```
// Статические конструкторы используются для инициализации статических переменных
public class Employee
{
...
        private static string CompName;

        static Employee()
        {
                CompName = "Intertech, Inc";
        }
...
}
```

Если нам потребуется использовать свойство Employee.Company, присваивать ему исходное значение не придется — статический конструктор сделает это за нас автоматически:

```
// Значение свойства ("Intertech, Inc") будет автоматически установлено
// через статический конструктор
public static int Main(string[] args)
{
...
        Console.WriteLine("These folks work at {0}", Employee.Company);
}
```

Подводя итоги этого раздела, можно отметить, что свойства классов C# используются для тех же самых целей, что и традиционные методы доступа и изменения значений. Главное преимущество свойств заключается в том, что пользователь может работать через них со внутренними данными, используя единственное имя (вместо двух разных имен методов).

Псевдоинкапсуляция: создание полей «только для чтения»

Помимо свойств только для чтения, в C# также предусмотрены поля, значения которых изменять нельзя. Как мы помним, поля (fields) — это открытые данные класса (переменные, объявленные с ключевым словом public). Обычно применение полей в рабочем приложении — не самая лучшая идея, поскольку поля беззащитны — им легко присвоить ошибочное значение и тем самым испортить внутреннее состояние объекта. Однако в C# предусмотрена возможность запретить любую возможность изменять значение поля, объявив его с ключевым словом readonly:

```
public class Employee
{
```

```
...
        // Поле только для чтения (его значение устанавливается конструктором):
        public readonly string SSNField;
}
```

Как, наверное, вы уже догадываетесь, любая попытка изменить значение такого поля приведет к ошибке компилятора.

Статические поля «только для чтения»

Статические поля, определенные как «только для чтения», также вполне имеют право на существование. Обычно они используются в тех ситуациях, когда вы хотите создать некоторое количество постоянных значений, связанных с определенным классом. Очень похожие задачи выполняют обычные константы, которые можно назвать родственниками статических полей только для чтения. Однако между такими полями и константами есть существенное различие: константа заменяется на свое значение уже в процессе компиляции, в то время как значения статических полей только для чтения вычисляются лишь в процессе выполнения программы.

Например, предположим, что у нас есть объект Car (автомобиль), который в процессе выполнения должен создавать объект Ti re (шины). Для этой цели вы можете применить класс Ti re, использовав в нем набор статических полей только для чтения:

```
// В классе Tire определен набор полей только для чтения
public class Tire
{
        public static readonly      Tire GoodStone = new Tire(90);
        public static readonly      Tire FireYear = new Tire(100);
        public static readonly      Tire ReadyLine = new Tire(43);
        public static readonly      Tire Blimpy = new Tire(83);

        private int manufactureID;

        public int MakeID
        {
                get { return manufactureID; }
        }

        public Tire (int ID)
        {
                manufactureID = ID;
        }
}
```

А вот пример применения статических полей только для чтения:

```
// Так можно использовать динамически создаваемые поля только для чтения
public class Car
{
        // Какая у меня марка шин?
        public Tire tireType = Tire.Blimpy;     // Возвращает новый объект Tire
```

Г

```
public class CarApp
{
```

```
    public static int Main(string[] args)
    {
        Car c - new Car();

        // Выводим на консоль идентификатор производителя шин
        // (в нашей случае - 83)
        Console.WriteLine("Manufacture ID of tires: {0}", c.tireType.MakeID);
        return 0;
    }
}
```

Поддержка наследования в C#

Теперь, когда вы уже умеете создавать хорошо инкапсулированные классы, обратимся к созданию в C# наборов взаимосвязанных классов при помощи наследования. Как уже говорилось, наследование — это один из трех основных принципов объектно-ориентированного программирования. Главная задача наследования — обеспечить повторное использование кода. Существует два основных вида наследования: классическое наследование (отношение «быть» — is-a) и включение-делегирование (отношение «иметь» — has-a). Мы начнем с рассмотрения средств C# для реализации классического наследования.

При установлении между классами отношений классического наследования вы тем самым устанавливаете между ними зависимость. Основная идея классического наследования заключается в том, что производные классы должны получать функциональность от базового класса-предка и дополнять ее новыми возможностями. Рассмотрим это на примере. Предположим, что в дополнение к нашему классу Employee мы определили еще два производных класса: класс SalesPerson (продавец) и класс Manager (администратор). В результате получится иерархия классов, которая представлена на рис. 3.9.

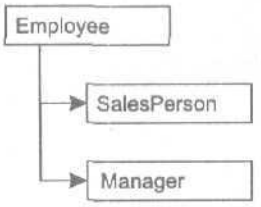


Рис. 3.9. Иерархия классов сотрудников

Как видно из рисунка, сотрудниками являются и продавец, и менеджер. В модели классического наследования базовые классы (в нашем случае Employee) используются для того, чтобы определить характеристики, которые будут общими для всех производных классов. Производные классы (такие как SalesPerson и Manager) расширяют унаследованную функциональность за счет своих собственных, специфических членов.

В C# указатель на базовый класс выглядит как двоеточие (:). Если переводить наши отношения наследования между категориями сотрудников на язык синтаксиса C#, то соответствующий код может выглядеть следующим образом:

```
// Добавляем в пространство имен Employees два новых производных класса
namespace Employees
{
public class Manager : Employee
{
        // Менеджерам необходимо знать количество имеющихся у них опчионов на акции
        private ulong numberOfOptions;
        public ulong NumbQpts
        {
                get { return numberOfptions; }
                set { numberOfOptions = value; }
        }
}

public class Salesperson : Employee
{
        // Продавцам нужно знать объем своих продаж
        private int numberOfSales;
        public int NumbSales
        {
                get { return numberOfSales; }
                set { numberOfSales = value: }
        }
}
```

В нашей ситуации оба производных класса расширяют функциональность базового класса за счет добавления свойств, уникальных для каждого класса. Поскольку мы использовали отношение «быть», то классы Salesperson и Manager автоматически унаследовали от класса Employee все открытые члены. Это несложно проверить:

```
// Создаем объект производного класса и проверяем его возможности
public static int Main(string[] args)
{
        // Создаем объект «продавец»
        Salesperson stan = new SalesPerson();

        // Эти члены унаследованы от базового класса Employee
        stan.EmpID = 100;
        stan.SetFullName("Stan the Man");

        // А это - уникальный член, определенный только в классе Salesperson
        stan.NumbSales = 42:

        return 0;
}
```

Конечно же, через объект производного класса нельзя напрямую обратиться к закрытым членам, определенным в базовом классе:

```
// Ошибка! Через объект производного класса нельзя обращаться к закрытым членам.
// определенным в базовом классе
SalesPerson stan = newSalesPersonn();
stan.currPay;
```

Работа с конструктором базового класса

Классы SalesPerson и Manager в том виде, в котором они существуют сейчас, могут быть созданы только при помощи конструкторов по умолчанию — просто потому,

что другие конструкторы не определены. Но, вполне возможно, нам удобнее создавать объекты этих классов, сразу же присваивая значения их внутренним данным, например:

```
// Создаем объект производного класса, используя собственный конструктор
Manager chucky = new Manager("Chucky", 92, 100000, "333-23-2322", 9000);
```

В коде, представленном выше, мы используем пользовательский конструктор. Однако, внимательно посмотрев на список параметров, которые мы ему передаем, мы можем заметить, что большинство из них предназначено для установки значений тех внутренних данных, которые унаследованы из базового класса. Наш вариант конструктора для производного класса может выглядеть так:

```
// При создании обьекта производного класса конструктор производного класса
// автоматически вызывает конструктор базового класса по умолчанию
public Manager(string fullName, int empID, float currPay, string ssn, ulong numbOfOpts)
{
        // Присваиваем значения уникальным данным нашего класса
        numberOfOptions = numbOfOpts;

        // Присваиваем значения данным, унаследованным от базового класса
        EmpID = empID;
        SetFullName(fullName);
        SSN = ssn;
        Pay = currPay;
}
```

Такое решение вполне допустимо, но с точки зрения производительности оно не является оптимальным. Все равно пользовательскому конструктору производного класса приходится вызывать конструктор базового класса (если иное не указано специально, будет вызван конструктор по умолчанию), который присвоит всем данным, за которые он «ответствен», безопасные значения по умолчанию. Только после этого в дело вступит пользовательский конструктор производного класса, который заново определит значения для унаследованных членов. Конечно, гораздо удобнее и эффективнее с точки зрения производительности (а иногда это и просто единственный возможный способ) — сразу вызвать нужный вариант пользовательского конструктора базового класса. В C# имеются для этого средства:

```
// Обратите внимание на использование ключевого слова base для вызова пользовательского
// конструктора базового класса
public Manager(string fullName, int empID, float currPay, string ssn, ulong numbOfOpts)
                        : base(fullName, empID, currPay, ssn)
{
        numberOfOptions = numbOfOpts;
}
```

Как мы видим, в объявлении нашего пользовательского конструктора производного класса появилось небольшое дополнение в виде двоеточия, следующего за ним ключевого слова base и перечня параметров в скобках. Все это означает, что будет вызван тот вариант конструктора базового класса, который принимает четыре параметра, и нам уже не потребуется присваивать значения унаследованным членам в конструкторе производного класса. С точки зрения производительности такое решение значительно эффективнее. Создание объекта класса SalesPerson при использовании этого приема выглядит практически идентично:

```
// При создании объекта производного класса часто бывает выгодный явно указать вызывае-
мый вариант конструктора базового класса
public SalesPerson(string fullName, int emplD. float currPay, ssn) : base(fullName,
empID, currPay, ssn)
{
        this.numberOfSales = numbOfSales;
}
```

Ключевое слово base можно использовать в любой ситуации, когда из производного класса необходимо обратиться к открытым или защищенным членам базового класса: применение этого ключевого слова не ограничено конструктором.

Можно ли производить наследование от нескольких базовых классов

Очень важно отметить, что в C# при создании производного класса в качестве базового класса можно указать только один класс. Это означает, что множественное наследование в C# запрещено. Разрешение множественного наследования порождает большое количество проблем, и именно поэтому при создании C# было принято решение от него отказаться. Однако программисты на C# имеют в своем распоряжении другую возможность: в C# можно производить один интерфейс от нескольких. Множественное наследование для интерфейсов не запрещено. Подробнее об этом будет рассказано в главе 4.

Хранение «семейных тайн»: ключевое слово protected

Как мы уже знаем, открытые члены класса (объявленные как public) могут быть доступны откуда угодно. Закрытые члены класса (объявленные как private) могут быть доступны только из того объекта, в котором они явно определены. Однако в C#, как и в других современных объектно-ориентированных языках программирования, предусмотрен еще один модификатор области видимости: protected (защищенное).

При создании защищенных переменных или защищенных методов класса эти члены будут доступны напрямую как из собственного класса, так и из всех производных классов. Если вы хотите, чтобы наши производные классы Salesperson и Manager получили доступ напрямую к данным, определенным в базовом классе Employee, эти данные в Employee должны быть определены следующим образом:

```
// Защищенные данные о состоянии
public class Employee
{
        // Производные классы смогут напрямую обращаться к этой информации.
        // Пользователи объектов класса - нет
        protected string fullName;
        protected int emplD;
        protected float currPay:
        protected string ssn:
...
}
```

С точки зрения пользователей объектов защищенные данные являются закрытыми (то есть protected для них будет эквивалентно private). Поэтому следующий код приведет лишь к сообщению об ошибке:

```
// Ошибка! Через экземпляр объекта напрямую к защищенным данным обратиться нельзя
Employee emp = new Employee();
emp.ssn  -  "111-11-1111";
```

Запрет наследования: классы, объявленные как sealed

Классическое наследование — это очень удобный способ определить набор характеристик для класса. Однако иногда встречаются ситуации, когда вы не просто не хотите использовать наследование, но желаете запретить производить наследование от какого-либо класса. Например, предположим, что в нашей иерархии сотрудников появился новый класс — PTSalesPerson (от part-time — продавцы на неполный рабочий день), который наследует существующему классу SalesPerson. Схема отношений между классами, представляющими сотрудников, теперь выглядит так, как показано на рис. 3.10.

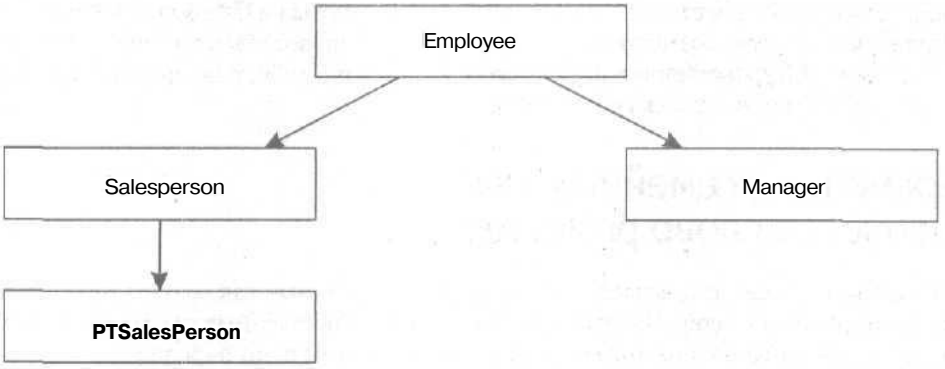


Рис. 3.10. Иерархия классов сотрудников в расширенном составе

Предположим, что во избежание недоразумений мы хотим запретить остальным разработчикам (и себе в том числе) производить какие-либо классы от PTSalesPerson. В C# для запрещения наследования от какого-либо класса предусмотрено ключевое слово sealed (переводится как «закрытый на ключ»):

```
// Запрещаем классу PTSalesPerson выступать в качестве базового для остальных классов
public sealed class PTSalesPerson : Salesperson
{
        public PTSalesPerson(string fullName, int empID, float currPay, string ssn, int
                        numbOfSales) : base(fullName, empID, currPay, ssn, numbOf Sales)
        {
                // Код конструктора
        }
        // Члены класса...
}
```

Поскольку мы объявили PTSalesPerson как закрытый, его уже нельзя использовать в качестве базовых для остальных классов. Подобный код приведет лишь к ошибке компилятора:

```
// Ошибка! PTSalesPerson не может выступать а качестве базового класса
// при наследовании
public class ReallyPTSalesPerson : PTSalesPerson
{
        ...
}
```

Чаще всего ключевое слово sealed применяется для обособленных служебных классов. Например, таким образом объявлен класс String пространства имен System. Поэтому не удастся создать класс, производный от System. String. Если вам все же крайне необходимо включить возможности System. String в свой собственный класс, в вашем распоряжении есть еще один способ — воспользоваться моделью включение-делегирование. Речь о ней идет ниже.

Применение модели включения-делегирования

Как уже говорилось, в объектно-ориентированных языках программирования используются две главные разновидности наследования. Первая из них называется классическим наследованием (моделью «быть» — is-a), и эта модель была рассмотрена в предыдущем разделе. Вторая разновидность — это модель включения-делегирования (модель «иметь» — has-a), и именно ей посвящен настоящий раздел. Для начала нам потребуется простой класс, представляющий автомобильный радиоприемник:

```
// Этот класс будет внутренним, включенным в другой класс - Car
public class Radio
{
        public Radio(){}

        public void TurnOn(bool on)
        {
                if(on)
                        Console.WriteLine("Jamming...");
                else
                        Console.WriteLine("Quiet time...");
        }
}
```

Теперь предположим, что вам потребовалось разработать программную модель автомобиля. Класс Car, который мы будет использовать для этой цели, должен содержать сведения об автомобиле (как мы его именуем, его скорость в настоящий момент и максимально допустимую скорость). Все эти данные можно задавать при помощи пользовательского конструктора. Определение класса Car может выглядеть следующим образом:

```
// Этот класс будет выступать в роли внешнего класса, класса-контейнера для Radio
public class Car
{
        private int currSpeed;
        private int maxSpeed;
        private string petName;
        bool dead:                      // Жива ли машина или уже нет

        public CarC)
        {
                maxSpeed = 100;
```

```
            dead - false;
        }

        public Car(string name, int max. int curr)
        {
            currSpeed - curr;
            maxSpeed - max;
            petName = name;
            dead = false;
        }

        public void SpeedUp (int delta)
        {
            // Если машина уже «мертва» (при превышении максимальной скорости).
            // то следует сообщить об этом
            if(dead)
            {
                Console.WriteLine(petName + " is out of order...");
            }
            else                    // Пока еще все нормально, увеличиваем скорость
            {
                currSpeed += delta;
                if(currSpeed >= maxSpeed)
                {
                    Console.WriteLine(petName + " has overheated...");
                    dead = true;
                }
                else
                    Console.WriteLine("\tCurrSpeed - " + currSpeed);
            }
        }
}
```

Сейчас в нашем распоряжении есть два независимых класса — Radio для автомобильного радиоприемника и Car для самого автомобиля. Понятно, что эти два класса должны взаимодействовать друг с другом и эти отношения желательно как-то зафиксировать. Однако вряд ли нам удастся применить в этом случае классическое наследование: трудно производить машину от радиоприемника или радиоприемник от машины. В этой ситуации, конечно, больше подойдет отношение включения-делегирования: пусть машина включает в себя радио и передает этому классу необходимые команды. В терминологии объектно-ориентированного программирования контейнерный класс (в нашем случае Car) называется родительским (parent), а внутренний класс, который помещен внутрь контейнерного (это, конечно, Radio), называется дочерним (child).

Помещение радиоприемника внутрь автомобиля влечет за собой внесение в определение класса Car следующих изменений:

```
// Автомобиль «имеет» (has-a) радио
public class Car
{
...
    // Внутреннее радио
    private Radio theMusicBox;
```

Г

Обратите внимание, что внутренний класс Radio был объявлен как private. С точки зрения инкапсуляции мы делаем все правильно. Однако при этом неизбежно

возникает вопрос: а как нам включить радио? Переводя на язык программирования — а как внешний мир будет взаимодействовать с внутренним классом? Понятно, что ответственность за создание объекта внутреннего класса несет внешний контейнерный класс. В принципе код для создания объектов внутреннего класса можно помещать куда угодно, но обычно он помещается среди конструкторов контейнерного класса:

```
// За создание объектов внутренних классов ответственны контейнерные классы
public class Car
{
...
        // Встроенное радио
        private Radio theMusicBox:

        public Car()
        {
                maxSpeed = 100;
                dead = false;
                // Объект внешнего класса создаст необходимые объекты внутреннего класса
                // при собственно» создании
                theMusicBox = new Radio();   // Если мы этого не сделаем. theMusicBox
                                             // начнет свою жизнь с нулевой ссылки
        }

        public Car(string name. int max. int curr)
        {
                currSpeed = curr:
                maxSpeed = max;
                petName = name;
                dead - fales:
                theMusicBox = newRadio();
        }
...
}
```

Произвести инициализацию средствами C# можно и так:

```
// Автомобиль «имеет» (has-a) радио
public class Car
{
...
        // Встроенное радио
        private Radio theMusicBox = new Radio();
...
}
```

Таким образом, радиоприемник внутри автомобиля у нас теперь создается: вместе с автомобилем. Однако вопрос о том, как именно можно включить этот радиоприемник, остался нерешенным. Ответ на него выглядит так: для того чтобы воспользоваться возможностями внутреннего класса, необходимо *делегирование* (delegation). Делегирование заключается в простом добавлении во внешний контейнерный класс методов для обращения ко внутреннему классу, Например:

```
// Во внешний класс добавляются дополнительные открытые методы и другие члены.
// которые обеспечивают доступ к внутреннему классу
public class Car
```

```
{
...
        public void CrankTunes(bool state)
        {
                // Передаем (делегируем) запрос внутреннему объекту
                theMusicBox.TurnOn(state);
        }
}
```

В приведенном ниже коде обратите внимание на то, что пользователь косвенно обращается к скрытому внутреннему объекту, даже не подозревая о том, что в недрах объекта `Car` существует закрытый (определенный как `private`) объект `Radio`:

```
// Выводим автомобиль на пробную поездку
public class CarApp
{
        public static int Main(string[] args)
        {
                // Создаем автомобиль (который, в свою очередь, создаст радио)
                Car c1;
                c:1 = new Car("SlugBug", 100, 10);

                // Включаем радио (запрос будет передан внутреннему рбъекту)
                c1.CrankTunes(true);

                // Ускоряемся
                for(int 1 = O; i < 10; i++)
                c1.SpeedUp(20);

                // Выключаем радио (запрос будет вновь передан внутреннему объекту
                c1.CrankTunes(false);
                return 0;
        }
}
```

Результат работы нашей программы представлен на рис. 3.11.

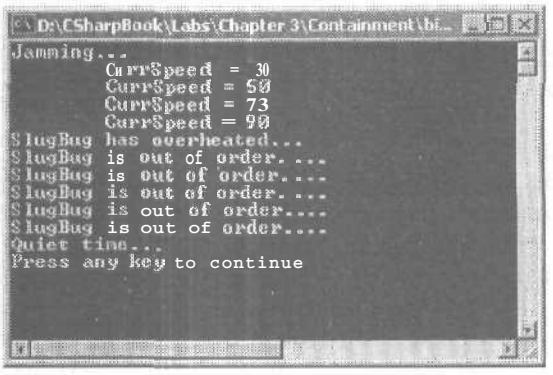


Рис. 3.11. Как работает радио внутри автомобиля

Код приложения Containtment можно найти в подкаталоге Chapter 3.

Определение вложенных типов

В C# вы можете определить тип непосредственно внутри другого типа. Такие типы называются вложенными (nested):

```
// В C# вы можете вкладывать друг в друга классы, интерфейсы и структуры
public class MyClass
{
        // Члены внешнего класса
        ...
        public class MyNestedClass
        {
                // Члены внутреннего класса
                ...
        }
}
```

Вряд ли представленный выше синтаксис требует каких-либо объяснений. Затруднения могут возникнуть лишь с вопросом: а зачем вообще может понадобиться вкладывать определения типов друг в друга? Как правило, вложенные типы — это исключительно вспомогательные типы, используемые только теми внешними типами, в которых они непосредственно определены. Обращение к вложенным типам напрямую из внешнего мира невозможно. Реально функциональность вложенных типов практически совпадает с функциональностью внутренних типов в модели включения–делегирования, за исключением того, что для вложенных типов обеспечивается более строгий контроль области видимости. В этом отношении применение вложенных типов помогает обеспечить еще более полное соответствие принципам инкапсуляции.

Для того чтобы продемонстрировать применение вложенных типов на практике, мы слегка изменим наше приложение Car таким образом, чтобы класс Radio стал вложенным для класса Car. При этом мы предполагаем, что к классу Radio из внешнего мира напрямую обратиться будет необязательно:

```
// Класс Radio вложен в класс Car. Все остальное - как в предыдущем приложении
public class Car : Object
{
...
        // К вложенному закрытому классу Radio нельзя обратиться из внешнего мира
        private class Radio
        {
                public Radio(){}
                public void TurnOn(bool on)
                {
                        if(on)
                                Console.WriteLine("Jamming...");
                        else
                                Console.WriteLine("Quiet time...");
                }
        }

        // Внешний класс может создавать экземпляры вложенных типов
        private Radio theMusicBox;
...
}
```

Обратите внимание, что класс Car имеет возможность создавать объекты любых вложенных в него типов. Также необходимо отметить, что вложенный класс объявлен как private. В C# вложенные классы могут объявляться и как private, и как public. Однако классы, которые объявлены в пространстве имен напрямую (то есть те классы, которые не вложены ни в какой другой класс), не могут быть объявлены как private.

Новый вариант приложения Car работает так же, как и предыдущий. Мы также не можем напрямую обратиться из внешнего мира к классу Radio:

```
// За пределами класса Car такая операция приведет только к ошибке!
Radio r - new Radio();
```

Код приложения Nested можно найти в подкаталоге Chapter 3.

Поддержка полиморфизма в C#

Предположим, что в базовом классе Employee определен метод GiveBonus() — поощрить:

```
// В классе Employee определен новый метод для поощрения сотрудников
public class Employee
{
...
        public void GiveBonus(float amount)
        {
                currPay += amount;
        }
...
}
```

Поскольку этот метод определен в базовом классе как public, вы теперь можете поощрять продавцов и менеджеров:

```
// Поощрения объектам производных классов
Manager chucky - new Manager("Chucky", 92. 100000, "333-23-2322", 9000);
chucky.GiveBonus(300);
chucky.DisplayStats();

SalesPerson fran - new SalesPerson("Fran", 93. 3000. "932-32-3232", 31);
fran.GiveBonus(200);
fran. DisplayStatsO;
```

Результат выполнения поощрения представлен на рис. 3.12.

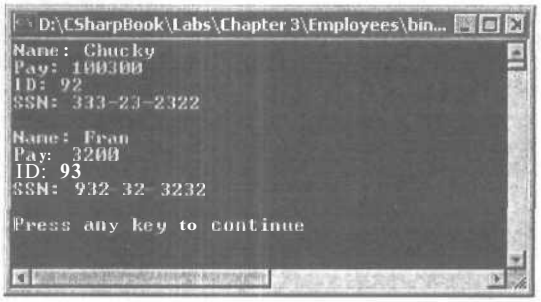


Рис. 3.12. Полиморфизм не используется; метод действует одинаково в отношении обоих производных классов

Проблема заключается в том, что унаследованный метод GiveBonus() пока работает абсолютно одинаково в отношении объектов обоих производных классов — и объектов SalesPerson, и объектов Manager. Однако, конечно, было бы лучше, чтобы для объекта каждого класса использовался свой уникальный вариант метода. Например, при поощрении продавцов можно учитывать их объем продаж. Менеджерам помимо денежного поощрения можно выдавать дополнительные опционы на акции. Поэтому задачу можно сформулировать так; «Как заставить один и тот же метод по-разному реагировать на объекты разных классов?»

Для решения этой задачи в C# предусмотрено понятие полиморфизма. Полиморфизм позволяет переопределять реакцию объекта производного класса на метод, определенный в базовом классе. Для реализации полиморфизма в нашем приложении мы воспользуемся ключевыми словами C#: virtual и override. Если базовый класс определяет метод, который должен быть замещен в производном классе, этот метод должен быть объявлен как виртуальный (конечно, при помощи ключевого слова virtual):

```
public class Employee
{
        // Для метода GiveBonusO предусмотрена реализация по умолчанию.
        // однако он может быть замещен в производных классах
        public virtual void GiveBonus(float amount)
        {
                currPay += amount:
        }
...
}
```

Если вы хотите переопределить виртуальный метод, необходимо заново определит ь этот метод в производном классе, использовав ключевое слово override:

```
public class Salesperson : Employee
{
        // На размер поощрения продавцу будет влиять объем его продаж
        public override void GiveBonusCfloat amount)
        {
                int salesBonus = 0:

                if(numberOfSales >= 0 && numberOfSales <=100)
                        salesBonus = 10;
                else if(numberOfSales >= 101 && numberOfSales <= 200)
                        salesBonus - 15;
                else
                        salesBonus  = 20:      // Для объема продаж больше 200

                base.GiveBonus (amount * salesBonus):
        }
...
}
public class Manager : Employee
{
        private Random r = new Random();

        // Помимо денег менеджеры также получают некоторое количество опционов
        // на акции
```

```
        public override void GiveBonus(float amount)
        {
            // Деньги: увеличиваем зарплату
            base.GiveBonus(amount);

            // Опционы на акции: увеличиваем их количество
            numberOfOptions += (ulong)r.Next(500);
        }
...
}
```

Обратите внимание, что в определении каждого из замещенных методов используется вызов этого же метода базового класса. Таким образом, нет необходимости заново определять всю логику замещенного метода в производном классе: вполне достаточно воспользоваться вариантом метода по умолчанию, определенном в базовом классе, дополнив его нужными вам действиями.

Метод `Employee.DisplayStats()` у нас также определен как virtual, и он замещен в производных классах таким образом, чтобы показывать текущий объем продаж, если речь идет о продавце, или количество опционов, имеющееся в настоящее время в распоряжении менеджера. Теперь, когда для каждого из производных классов определены собственные варианты этих двух методов, объекты разных классов ведут себя по-разному:

```
// Улучшенная система поощрений!
Manager chucky = new Manager("Chucky", 92, 100000, "333-23-2322", 9000);
chucky.GiveBonus(300);
chucky.DisplayStats();

Salesperson fran = new SalesPerson("Fran", 93, 3000, "932-32-3232", 31);
fran.GiveBonus(200);
fran.DisplayStats();
```

Результат раздачи поощрений представлен на рис. 3.13.

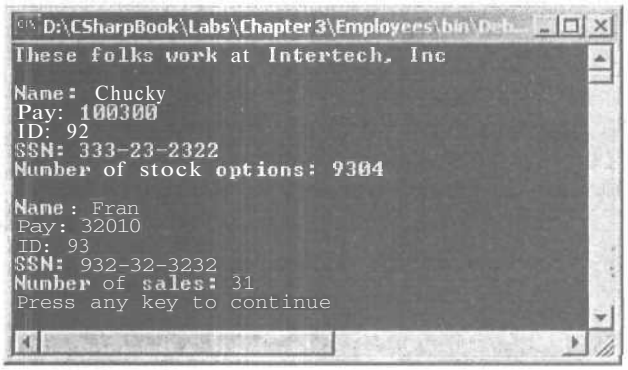


Рис. 3.13. Благодаря полиморфизму мы смогли реализовать более удобную систему поощрений

Мы с вами научились использовать полиморфизм для переопределения поведения производных классов. Однако, как, наверное, вы догадываетесь, возможности полиморфизма этим далеко не исчерпываются.

Абстрактные классы

В настоящее время базовый класс Employee выполняет в нашей программе понятные и логичные функции: он обеспечивает производные классы набором защищенных переменных, а также предоставляет реализации по умолчанию двух виртуальных методов— GiveBonus() и DisplayStats(), замещенных в производных классах. Однако в нашей программе до сих пор существует потенциальный источник ошибок: мы можем создавать объекты базового класса Employee:

```
// А это кто такой?
Employee X = new Employee();
```

«Просто сотрудников» у нас быть не должно — каждой из возможных категорий сотрудников у нас соответствует производный класс. Поэтому вполне логичным будет просто запретить создание объектов класса Employee. В C# для этого достаточно объявить класс абстрактным:

```
// Объявляем класс Employee абстрактным, запрещая создание объектов этого класса
abstract public class Employee
{
        // Открытые интерфейсы и внутренние данные класса
}
```

Теперь при попытке создания объекта класса Employee компилятор будет выдавать сообщение об ошибке:

```
// Ошибка! Нельзя создавать экземпляры абстрактного класса
Employee X = new Employee();
```

Принудительный полиморфизм: абстрактные методы

После того как мы объявили класс абстрактным, можно определить в нем любое количество абстрактных методов. Абстрактные методы — это аналоги чистых виртуальных функций C++: они позволяют определить в базовом классе методы без реализации по умолчанию. Все виртуальные методы обязательно должны быть замещены в производных классах.

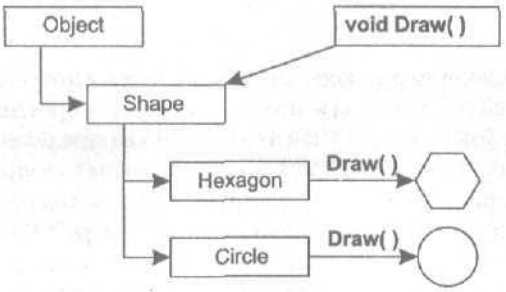


Рис. 3.14. Иерархия классов геометрических фигур

Скорее всего, вы спросите: «А зачем это нужно?» Для ответа на этот вопрос мы вернемся к иерархии геометрических фигур, с которой мы уже имели дело в этой главе (рис. 3.14).

Как и в случае с классом Employee, желательно явным образом запретить создание объектов класса Shape. Это можно сделать следующим образом:

```
namespace Shapes
{
public abstract class Shape
{
        // Пусть каждый объект-геометрическая фигура получит у нас дружеское прозвище:
        protected string petName:

        // Конструкторы
        public Shape() {petName = "NoName";}
        public Shape(string s) {petName = s;}

        // Метод Draw() сбъявлен как виртуальный и может быть замещен
        public virtual void DrawO
        {
                Console.WriteLine("Shape.Draw()");
        }
        public string PetName
        {
                get {return petName;}
                set {petName = value;}
        }
)
// В классе Circle метод DrawC) НЕ ЗАМЕЩЕН
public class Circle : Shape
{
        public Circle() {}
        public Circle(string name): base(name) {}
}

// В классе Hexagon метод DrawO ЗАМЕЩЕН
public class Hexagon : Shape
{
        public HexagonO {}
        public Hexagon(string name) : base(name) {}
        public overrride void DrawO
        {
                Console.WriteLine("Drawing {0} the Hexagon", PetName);
        }
}
}
```

Обратите внимание, что в классе Shape определен виртуальный метод Draw(). Как мы видим, виртуальные методы можно замещать в производных классах при помощи ключевого слова override (как это сделано в определении класса Hexagon). Однако в C# виртуальные методы можно и не замещать в производных классах (например, в определении класса Circle виртуальный метод Draw() базового класса остался незамещенным). При этом в случае вызова метода Draw() для объекта класса Hexagon будет вызван уникальный вариант этого метода для класса Hexagon, а если мы вызовем тот же метод для объекта класса Circle, этот метод будет выполнен в соответствии со своим определением в базовом классе:

```
// В объекте Circle реализация базового класса для DrawO не замещена
public static int Main(string[] args)
{
        // Создаем и рисуем шестиугольник
```

```
Hexagon hex = new Hexagon("Beth");
hex.Draw();

Circle cir - new Circle("Cindy");
// М-мм! Придется использовать реализацию Draw() базового класса
cir.Draw();
```

Г

Результат работы этой программы представлен на рис. 3.15.

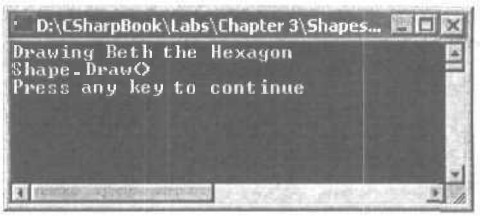


Рис. 3.15. Виртуальные методы замещать не обязательно

Если нам необходимо гарантировать, что каждый производный класс обязательно заместит метод Draw(), то мы должны объявить метод Draw() в базовом классе Shape абстрактным. Абстрактные методы в C# работают так же, как чистые виртуальные функции в C++ — для них даже не надо указывать реализацию по умолчанию:

```
// Каждая геометрическая фигура теперь ОБЯЗАНА самостоятельно определять
// метод Draw()
public abstract class Shape
{
...
        // Метод DrawO теперь определен как абстрактный (обратите внимание
        // на точку с запятой)
        public abstract void DrawO;
        public string PetName
        {
                get {return petName;}
                set {petName - value;}
        }
}
```

Теперь мы обязаны определить метод Draw() в классе Circle:

```
// Если мы не заместим в классе Circle абстрактный метод Draw(), класс Circle будет
// также считаться абстрактным и мы не сможем создавать объекты этого класса!
public class Circle : Shape
{
        public CircleO {}
        public Circle(string name): base(name) {}

        // Теперь метод DrawO придется замещать в любом производном непосредственно
        // от Shape классе
        public override void DrawC)
        {
                Console.WriteLine("Drawing {0} the Cricle", PetName):
        }
```

Теперь мы можем воспользоваться полной мощью полиморфизма C#, когда конкретная реализация метода будет выбираться автоматически в зависимости от того, для объекта какого класса был вызван этот метод:

```
// Создаем массив объектов разных геометрических фигур
public static int Ma1n(str1ng[] args)
{
        // Массив фигур
        Shape[] s ={new Hexagon(), new Hexagon("Freda"),
                    new Circle(); new Circle("JoJo")};

        // Проходим с помощью цикла по всем элементам массива и просим нарисовать
        // каждый объект
        for(int 1 = 0: i < s.Length; i++)
              s[i].Draw();
...
}
```

Результат выполнения программы представлен на рис. 3.16.

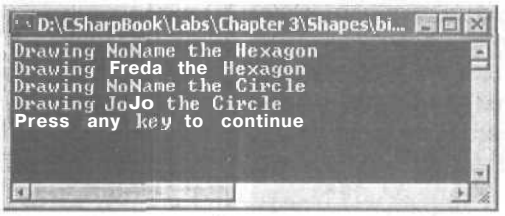


Рис. 3.16. Абстрактные методы ДОЛЖНЫ быть замещены

Этот пример очень хорошо иллюстрирует суть полиморфизма. Как мы помним, объекты абстрактных классов создавать невозможно. Однако вы вполне можете хранить ссылки на любой производный класс, используя для этого переменную абстрактного класса. В нашем примере при прохождении по элементам массива геометрических фигур нужный вариант метода выбирается непосредственно в момент выполнения программы в зависимости от класса, к которому принадлежит объект.

Контроль версий членов класса

В C# существует еще один прием работы с методами в производных классах, который можно противопоставить замещению методов. Этот прием называется сокрытием методов (method hiding). Рассмотрим его на примере.

Предположим, что вы создаете новый класс с именем Oval (для новой геометрической фигуры нашей иерархии — овала). Пусть у нас Oval является производным классом от класса Circle. Тогда наша иерархия геометрических фигур будет выглядеть так, как показано на рис. 3.17.

Теперь предположим, что в классе Oval также определен метод Draw(). Однако представим, что, в отличие от отношений между реализациями метода Draw() в базовом классе, мы хотим запретить любое наследование логики Draw() между методами — фактически разорвать отношения наследования на уровне единственного метода. C# предлагает для этого случая средство, которое называется контролем

версий (versioning). Для того чтобы **им** воспользоваться, достаточно определить в классе Oval метод Draw() с ключевым словом new:

```
// Класс Oval наследует классу Circle, однако сокрывает унаследованный метод Draw()
public class Oval : Circle
{
        public Oval() {base.PetName = "Joe";}

        // Скрываем любые реализации DrawO базовых классов
        public new void DrawO
        (
                // Специфичный для Oval алгоритм DrawO
        }
}
```

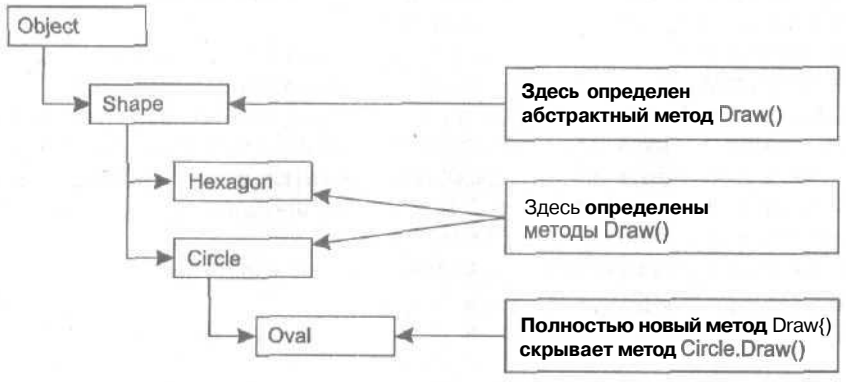


Рис. 3.17. Для метода Draw() класса Oval мы применяем возможности контроля версий членов класса

Поскольку в классе Oval для метода Draw() у нас использовано ключевое слово new, мы гарантируем, что при создании объекта класса Oval и вызове для него метода Draw() будет вызвана именно та реализация этого метода, которая определена в классе Oval. Ключевое слово new разрывает отношения между абстрактным методом Draw(), определенным в базовом классе Shape, и методом Draw() в классе Oval.

```
// Будет вызван метод DrawO. определенный в классе Oval
Oval o = new OvalO:
o.DrawO;
```

Однако если вам все же потребуется вызвать вариант метода для базового класса, это тоже можно сделать — с помощью явного приведения типов:

```
// Будет вызван метод DrawO, определенный в классе Circle
Oval o = new Oval();
((Circle)o).Draw();  // Приводим о к базовоиу классу
```

На этом примере можно убедиться, что сокрытие методов — это не просто интересное упражнение при проектировании классов, но достаточно полезное средство, часто используемое в реальных проектах. Особенно часто оно применяется в тех ситуациях, когда вам необходимо произвести класс от базового класса, определенного в другой сборке .NET. Предположим, что в базовом классе у нас опреде-

лен метод Draw(), несовместимый с нашей собственной реализацией метода Draw(). Лучший способ запретить пользователям объектов обращаться к реализации метода в базовом классе — как раз воспользоваться сокрытием методов, объявив в производном классе нашу собственную реализацию метода с ключевым словом new.

Код приложения Shapes можно найти в подкаталоге Chapter 3.

Приведение типов в C#

К настоящему моменту мы создали уже не одну развитую иерархию типов. При этом C# позволяет приводить (cast) один тип к другому, то есть осуществлять преобразование объектов одного типа в объекты другого типа. Рассмотрению правил приведения типов в C# и посвящен этот раздел.

Рассмотрим приведение типов на простом примере. Вспомним нашу иерархию классов сотрудников. Конечно же, на самой вершине этой иерархии стоит класс System.Object — в C# все типы производятся от этого класса. Можно сказать, используя терминологию классического наследования, что все типы являются is-a-объектами. Кроме того, в нашей иерархии существуют и другие отношения классического наследования. Например, PTSalesPerson (продавец на неполный рабочий день) является is-a-продавцом Salesperson и т. д.

Все указанные ниже операции приведения типов допустимы:

```
// Класс Manager - производный от System.Object
object o = new Manager("Frank Zappa", 9, 40000, "111-11-1111", 5);

// Класс Manager - производный от Employee
Employee e = new Manager("MoonUnit Zappa", 2, 20000, "101-11-1321", 1);

// Класс PTSalesPerson - производный от SalesPerson
Salesperson sp = new PTSalesPerson("Jill", 84, 100000, "111-12-1119", 90);
```

Первый закон приведения типов звучит так: если один класс является производным от другого, всегда безопасно ссылаться на объект производного класса через объект базового класса. В результате мы можем использовать в C# весьма мощные программные конструкции. Например, если у нас определен метод для увольнения сотрудника:

```
public class TheMachine
{
        public static void FireThisPerson(Employee e)
        {
                // Удаляем сотрудника из базы данных
                // Отбираем у него ключ и точилку для карандашей
        }
}
```

В соответствии с правилами приведения типов мы можем передавать методу FireThisPerson() объект как самого типа Employee, так и любого производного от Employee типа:

```
// Производим сокращение персонала
TheMachine.FireThisPerson(e);
TheMachine.FireThisPerson(sp);
```

Этот код будет выполнен без ошибок, поскольку здесь производится неявное приведение от базового класса (Employee) к производному. Однако что, если вы также хотите уволить объект класса Manager (который в настоящее время хранится через ссылку на объект базового класса)? Если вы попробуете передать ссылку на объект (типа System.Object) нашему методу FireThisPerson(), то вы получите сообщение об ошибке компилятора:

```
// Класс Manager - производный от System.Object, поэтому мы имеем право провести
// следующую операцию приведения:
object o - new Manager("Frank Zappa", 9. 40000,, "111-11-1111". 5);
...
TheMachine.FireThisPerson(o); // Ошибка компилятора!
```

Причина ошибки кроется в определении метода F1 reThisPerson(), который принимает объект типа Employee. Чтобы этой ошибки не возникало, нам необходимо явно привести объект базового класса System.Object к производному типу Empl oyee (учитывая происхождение нашего объекта o, это вполне возможно):

```
// Здесь будет ошибка - вначале нужно провести явное приведение типов:
// FireThisPersonO

// А вот так проблем не возникнет:
FireThisPerson((Employee)o);
```

Приведение числовых типов

Приведение числовых типов подчиняется примерно тем же правилам, что и приведение классов. Если вы пытаетесь привести «больший» числовой тип к «меньшему» (например, int — в byte), необходимо провести явное преобразование:

```
int x = 30000;
byte b = (byte)x;     // Возможна потеря данных...
```

Нас можно поздравить: теперь мы уже умеем создавать сложные иерархии пользовательских типов в C#. В главах 4 и 5 будут рассмотрены дополнительные приемы проектирования классов. Но прежде чем перейти к ним, мы должны рассмотреть еще два аспекта, тесно связанных с проектированием классов: обработку ошибок и управление памятью.

Обработка исключений

В течение множества лет обработка ошибок превращалась разработчиками, использующими Windows, в сложную смесь различных приемов. Множество программистов реализовывало свою собственную логику обработки ошибок в рамках одного-единственного конкретного приложения. Например, команда разработчиков могла определить свой набор констант для представления известных условий возникновения ошибок и использовать их как значения, возвращаемые методами, Помимо этого в Windows API было определено большое количество кодов ошибок, которые должны были обрабатываться при помощи #define, HRESULTS и множества прочих средств. Многие COM-разработчики использовали набор стандартных COM-интерфейсов для возвращения значимой информации об ошибках клиентам COM.

Очевидная проблема всех этих приемов обработки ошибок заключается в том, что все они — разные. Каждый прием привязан к конкретной технологии, конкретному языку или конкретному проекту. Чтобы наконец навести порядок во всем этом, .NET предлагает единую технику для обнаружения ошибок времени выполнения и передачи сообщений о них: это — структурированная обработка исключений (Structured Exception Handling, SEH).

Преимущества этого метода заключаются в том, что в распоряжение всех разработчиков предоставляется единый и хорошо продуманный подход к обработке ошибок, который к тому же является общим для всех языков .NET. Таким образом, разработчик, использующий C#, будет реализовывать обработку ошибок точно так же, как программист, использующий VB.NET или MC++, и все остальные разработчики, использующие платформу .NET. Разработчики также получают дополнительную возможность генерировать и перехватывать исключения между двоичными файлами, AppDomains (о них будет сказано — в главе 6) и компьютерами в независимом от языка стиле.

Для того чтобы понять, как применять исключения в C#, в первую очередь необходимо осознать, что исключения в C# — это объекты. Все системные и пользовательские исключения в C# производятся от класса System.Exception (который, в свою очередь, производится от класса System. Object). В табл. 3.1 представлен перечень наиболее интересных свойств класса Exception.

Таблица 3.1. Главные члены класса System.Exception

Свойство	Назначение
HelpLink	Это свойство возвращает URL файла справки с описанием ошибки
Message	Это свойство (только для чтения) возвращает текстовое описание соответствующей ошибки. Само сообщение об ошибке устанавливается как параметр конструктора
Source	Возвращает имя объекта (или приложения), которое сгенерировало ошибку
StackTrace	Это свойство (только для чтения) возвращает последовательность вызовов, которые привели к возникновению ошибки
InnerException	Это свойство может быть использовано для сохранения сведений об ошибке между сериями исключений

Генерация исключения

Чтобы продемонстрировать использование System.Exception, мы обратимся к классу Car, который уже использовался в этой главе, а точнее, к его методу SpeedUp(). Вот текущая реализация этого метода:

```
// В настоящее время SpeedUp()выводит сообщения об ошибках прямо на системную консоль
public void SpeedUp(int delta)
{
        // Если машины больше нет. сообщить об этом
        if(dead)
        {
                Console.WriteLine(petName + " is out of order...");
```

```
        }
        else   // Еще жива, можно увеличивать скорость
        {
                currSpeed += delta;
                if(currSpeed >= maxSpeed)
                {
                        Console.WriteLine(petName + " has overheated...");
                        dead = true;
                }
                else
                        Console.WriteLine("\tCurrSpeed = " + currSpeed);
        }
}
```

Для наших целей мы переделаем метод SpeedUp() таким образом, чтобы при попытке ускорить уже вышедший из строя автомобиль (dead = = true) генерировалось исключение. Прежде всего вам потребуется создать и настроить новый объект класса Exception (исключение). Для передачи этого объекта используется ключевое слово throw. Вот пример:

```
// При попытке ускорить вышедший из строя автонобиль будет сгенерировано исключение
public void SpeedUp(int delta)
{
        if (dead)
                throw new Exception("This car is already dead");
        else
        {
                ...
        }
}
```

Прежде чем выяснить, как вызвать это исключение, необходимо отметить еще несколько моментов.

Во-первых, при создании пользовательского класса только мы сами принимаем решения о том, когда будут возникать исключения. Здесь мы создали исключение таким образом, что оно будет сгенерировано всякий раз при применении метода SpeedUp() к машине в нерабочем состоянии (dead = = true), принудительно прекращая выполнение этого метода.

Конечно, вы можете изменить метод SpeedUp() таким образом, что он будет восстанавливаться автоматически, без генерации каких-либо исключений. Как правило, исключения должны генерироваться только тогда, когда выполнение какого-либо метода должно быть немедленно прервано. При проектировании класса одним из самых важных моментов является принятие решений о том, когда должны генерироваться исключения.

Во-вторых, необходимо помнить, что в библиотеках среды выполнения .NET уже определено множество готовых исключений, которые можно и нужно использовать. Например, в пространстве имен System определены такие важные исключения, как ArgumentOutOfRangeException, IndexOutOfRangeException, StackOverflowException и многие другие. В прочих пространствах имен определены свои исключения, относящиеся к тем областям, за которые отвечает соответствующее пространство имен. В пространстве имен System.Drawing.Printing определены исключения, которые могут возникнуть в процессе вывода на печать, в System.IO — исключения ввода-вывода и т. п.

Перехват исключений

Метод SpeedUp() готов сгенерировать исключение, однако кто перехватит это исключение и будет на него реагировать? Ответ прост: если мы создали код, генерирующий исключение, у нас должен быть блок try/catch для перехвата этого исключения. Этот блок может выглядеть, к примеру, так:

```
// Безопасно разгоняем автонобиль
public static 1nt Main(string[] args)
{
        // Создаем автомобиль
        Car buddha - new CarC'Buddha". 100, 20);

        // Пытаемся прибавить газ
        try
        {
                for(int i - 0; i < 10; i++)
                {
                        buddha.SpeedUp(10);
                }
        }
        catch(Exception e)    // Выводим сообщение и трассируем стек
        {
                Consol e.WriteLi ne C e.Message);
                Console.WriteLine(e.StackTrace);
        }

        return 0;
}
```

Результат выполнения нашей программы представлен на рис. 3.18.

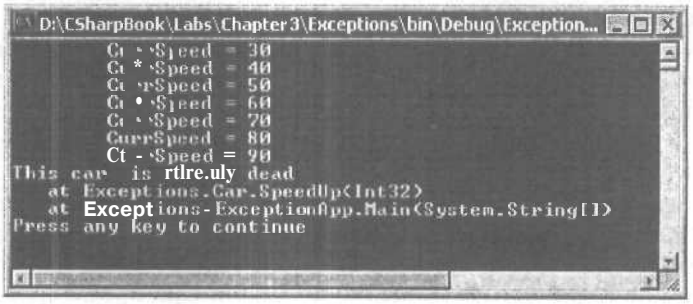


Рис. 3.18. Применение структурированной обработки исключений

В сущности, блок try — это отрезок кода, во время выполнения которого происходит отслеживание возникающих исключений. При возникновении исключения выполнение кода, определенного в блоке try, прерывается, и начинает выполняться код, определенный в ближайшем следующем блоке catch. Если же в процессе выполнения кода в блоке try исключений так и не возникло, блок catch полностью пропускается и выполнение программы идет дальше согласно ее внутренней логике.

Обратите внимание, что в нашем случае мы явно указали в блоке catch тип исключения, с которым мы рассчитываем встретиться. Однако C# (как и все осталь-

ные языки программирования .NET) позволяет настроить блок catch так, чтобы он реагировал на любые исключения, вне зависимости от их типа. Такой блок catch выглядит следующим образом:

```
// Захват всех исключений без разбора
catch
{
        Console.WriteLine("Something bad happened...");
}
```

Конечно же, подобная обработка исключений — это не лучший способ получить полную информацию о том, что произошло. C# предоставляет в ваше распоряжение мощные средства для работы с пользовательскими и системными исключениями.

Создание пользовательских исключений, первый этап

Несмотря на то что мы вполне можем ограничиться в нашем приложении только перехватом объектов класса System.Exception, часто бывает удобно создать свой собственный класс-исключение с необходимыми нам членами. Например, предположим, что мы создаем пользовательское исключение для знакомой нам ситуации с разгоном вышедшего из строя автомобиля. Первое, что нам нужно сделать, — определить новый класс, производный от System.Exception (по умолчанию для классов — пользовательских исключений используется суффикс Exception). Далее мы можем определить все необходимые нам свойства, методы и поля, которые будут использоваться внутри блока catch. Мы также можем: заместить любые виртуальные члены, определенные в базовом классе System.Exception:

```
// Это пользовательское исключение более подробно описывает ситуацию выхода машины
// из строя
public class CarIsDeadException : System.Exception
{
        // С помощью этого исключения мы сможем получить имя несчастливой машины
        private string carName;
        public CarIsDeadException(){}
        public CarIsDeadException(string carName)
        {
                this.carName = carName;
        }

        // Замещаем свойство Exception.Message
        public override string Message
        {
                get
                {
                        string msg = base.Message;

                        if(carName != null)
                                msg += carName + " has bought the farm...";

                        return msg;
                }
        }
}
```

Теперь наш класс-исключение CarIsDeadException содержит закрытую переменную carName для хранения информации об имени машины, для которой было сгенерировано исключение. Мы также добавили в класс два конструктора и заместили свойство Message таким образом, чтобы включить в описание исключения имя машины. Синтаксис генерации этого исключения очевиден:

```
// Генерируем пользовательское исключение
public void SpeedUp(int delta)
{
        // Если машина вышла из строя, сообщаем об этом
        if(dead)
        {
                // Генерируем исключение
                throw new CarIsDeadException(this.petName);
        }
        else    // Машина еще жива, можно разгоняться
        {
                currSpeed += delta;
                if(currSpeed >= maxSpeed)
                {
                        dead = true;
                }
                else
                        Console.WriteLine("\tCurrSpeed = {0}", currSpeed);
        }
}
```

Перехват этого исключения также не представит проблемы:

```
try
{
        ...
}
catch(CarIsDeadException e)
{
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);
}
```

Результат работы этого приложения представлен на рис. 3.19.

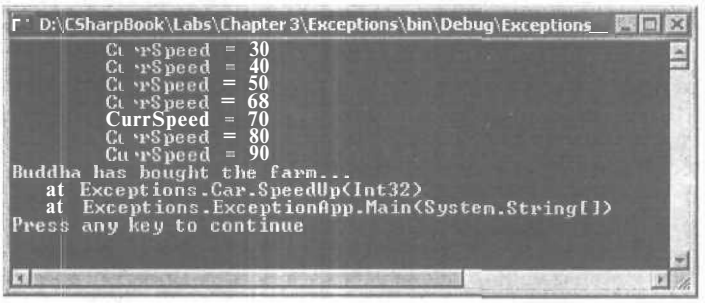


Рис. 3.19. Перехват пользовательского исключения

В нашем случае, в принципе, можно было бы обойтись и без создания класса пользовательского исключения. Как мы увидим чуть ниже, можно было бы просто

присвоить требуемое значение свойства Message системному классу исключения через его конструктор. Как правило, смысл создавать пользовательские исключения есть только тогда, когда ошибка тесно связана с пользовательским классом, в котором она возникла (например, класс File может генерировать ошибки, возникающие при работе с файлами).

Создание пользовательских исключений, второй этап

Наш пользовательский класс CarIsDeadException для выдачи сообщения об ошибке замещает свойство Message. Кроме того, в этом классе также предусмотрен конструктор, принимающий имя автомобиля, который стал причиной возникновения исключения. Мы можем создать любой пользовательский класс-исключение, удовлетворяющий нашим потребностям: это право разработчика. Однако в этом случае класс-исключение CarlsDeadException при тех же возможностях может выглядеть проще:

```
public class CarIsDeadException : System.Exception
{
        // Конструкторы для создания пользовательского сообщения об ошибке
        public CarIsDeadException(){}

        public CarIsDeadException(string message)
                : base(message){}

        public CarIsDeadException(string message. Exception innerEx)
                : base(message, innerEx){}
}
```

Обратите внимание, что в этом варианте нам не нужна строковая переменная для хранения имени машины и мы не замещаем свойство Message. Все, что нам нужно — это просто передать информацию членам базового класса. При генерации исключения мы передаем необходимую информацию как параметр конструктора (результат выполнения программы будет таким же, как и раньше):

```
SpeedUp(int delta)
{
        ...

        // Если машина вышла из строя - сообщить об этом
        if(dead)
        {
                // Передаем имя машины и сообщение как аргументы конструктора
                throw new CarIsDeadException(this.petName + " has bought the farm!");
        }
        else   // Машина пока жива, можно разгоняться
        {
                ...
        }
}
```

Такое решение нашего пользовательского класса-исключения, конечно, является более удачным — по крайней мере ввиду отсутствия лишних переменных и замещенных членов.

Обработка нескольких исключений

В наиболее простой форме одному блоку try соответствует один блок catch. Однако в реальных проектах часто возникают ситуации, когда нам необходимо отслеживать возникновение не одного, а нескольких исключений. Например, предположим, что наш метод SpeedUp() будет генерировать одно исключение, когда мы пытаемся разогнать вышедший из строя автомобиль (эту ситуацию мы уже разбирали), и другое — когда мы передаем этому методу неподходящие параметры (например, любое число меньше нуля).

```
// Проверка параметров на соответствие условиям
public void SpeedUp(int delta)
{
        // Ошибка в принимаемом параметре? Генерируем системное исключение
        if(delta < 0)
        throw new ArgumentOutOfRangeException("Must be greater than zero");

        // Если машина вышла из строя - сообщить об этом
        if(dead)
        {
                // Генерируем исключение CarIsDeadException
                throw new CarIsDeadException(this.petName + " has bought the farm!");
        }

        ...
}
```

Код для вызова исключений может выглядеть следующим образом:

```
// Теперь мы готовы перехватить оба исключения
try
{
        for(int i = 0; i < 10; i++)
                buddha.SpeedUp(10);
}
catch(CarIsDeadException e)
{
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);
}
catch(ArgumentOfRangeException e)
{
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);
}
```

Блок finally

После блока try/catch в C# может следовать необязательный блок finally. Этот блок выполняется всегда, вне зависимости от того, сработало исключение или нет. Его главное назначение — гарантировать, что ресурсы, которые могут быть открыты потенциально опасным методом, будут обязательно освобождены. Например, представим себе, что наша задача — сделать так, чтобы радио в автомобиле выключалось всегда при выходе из программы (метода Main()), вне зависимости от того, возникли или нет какие-нибудь ошибки в процессе выполнения:

```
// Используем блок finally для закрытия всех ресурсов
public static int Main(string[] args)
{
        Car buddha = new Car("Buddha", 100, 20);
        buddha.CrankTunes(true);

        // Давим на газ
        try
        {
                // Разгоняем машину...
        }
        catch(CarIsDeadException e)
        {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
        }
        catch(ArgumentOutOfRangeException e)
        {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
        }
        finally
        {
                // Этот блок будет выполнен всегда - вне зависимости от того,
                // произошла ошибка или нет
                buddha.CrankTunes(false);
        }
        return 0;
}
```

В нашей программе радио в автомобиле будет выключено всегда, вне зависимости от возникновения каких-либо исключений — потому, что мы поместили соответствующий код в блок f1 па 11y. Конечно, с помощью блока f1 па 11yможно не только «выключать радио» — в реальных проектах этот блок используется для освобождения памяти, закрытия файла, отключения от источника и данных и выполнения прочих операций, связанных с корректным завершением программы.

Код приложения Exceptions можно найти в подкаталоге Chapter 3.

Последние замечания о работе с исключениями

Как и все остальные средства обработки ошибок, исключения .NET не могут быть проигнорированы. Скорее всего, при чтении предыдущих разделов у вас уже возник вопрос: а что будет, если мы сгенерируем исключение, которое не будет перехватываться? Например, предположим, что у нас генерируется исключение CarIsDeadException, а соответствующий ему блок catch в программе отсутствует. Как правило, результатом необработанного исключения становятся смущающие конечного пользователя диалоги, подобные представленному на рис. 3.20.

Надеюсь, мне удалось убедить вас в том, что исключения в приложениях следует перехватывать. Осталось разобраться еще с одним важным моментом — а что делать с захваченным исключением? На этот вопрос придется отвечать вам как разработчику приложения. В нашем учебном примере с автомобилем результатом захвата исключения становится вывод на системную консоль определенного нами

сообщения и информации стека. В реальном приложении, к примеру, может происходить освобождение ресурсов и запись информации в файл журнала. Создание исключения и его перехват — это лишь общая схема, а реальное содержание, которым оно будет наполнено, зависит только от нас.

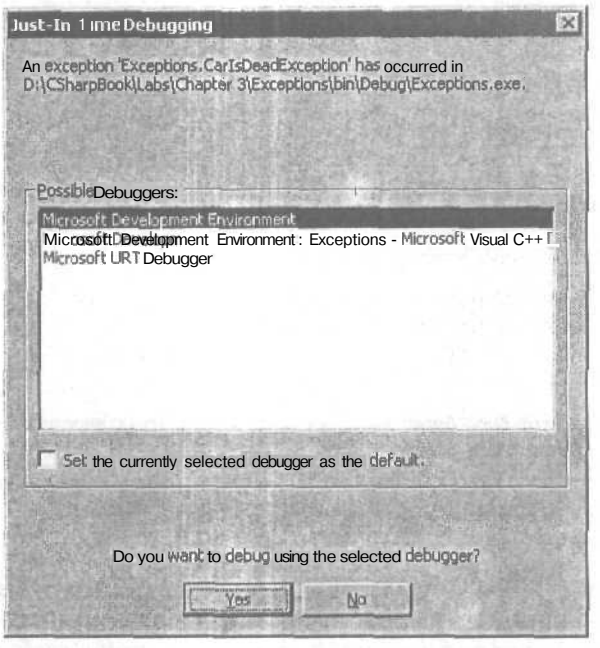


Рис. 3.20. В программе встретилось не перехваченное исключение

Будьте осторожны и не допускайте бесконечной генерации ошибок. Обычно такая ситуация возникает, когда ключевое слово throw помещается в блок catch:

```
try
[ // Код для ускорен/я автомобиля... }
catch(CarIsDeadException e)
{
        // Код для реакции на захваченное исключение
        // В этом коде мы генерируем то же самое исключение. Конечно, при необходимости
        // мы ножен генерировать и другое исключение
        throw e;
}
```

Наконец, отметим еще один важный момент. Необходимо принять за правило, что исключения должны генерироваться только тогда, когда возникает действительно непоправимая ситуация. Если у вас есть возможность выйти из положения с помощью обычной логики приложения, лучше постараться обойтись без исключения. В свете всего этого наше исключение `CarIsDeadException` вряд ли бы использовалось в реальном приложении. В главе 5 рассмотрен новый вариант метода `SpeedUp()`, в котором пользовательское исключение используется для более соответствующих его предназначению задач.

Жизненный цикл объектов

В C# общий принцип управления памятью формулируется очень просто: для создания объекта в области «управляемой кучи» (managed heap) используется ключевое слово new. Среда выполнения .NET автоматически удалит объект тогда, когда он больше не будет нужен. Правило в целом вполне понятно, однако возникает один дополнительный вопрос: а как среда выполнения определяет, что объект больше не нужен? Короткий (то есть неполный) ответ гласит, что среда выполнения удаляет объект из памяти, когда в текущей области видимости больше не остается активных ссылок на этот объект. Например:

```
// Создаем локальный объект класса Car
public static int Main(string[] args)
{
        // Помещаем объект класса Car в управляемую кучу
        Car c3 = new Car("Viper", 200, 100);
        ...
        return 0;
}   // Если c3 - единственная ссылка на этот объект, то начиная с этого момента
    // он может быть удален
```

Предположим, что в вашем приложении создано (размещено в оперативной памяти) три объекта класса Car. Если в управляемой куче достаточно места, мы получим три активные ссылки — по одной на каждый объект в оперативной памяти. Каждая такая активная ссылка на объект в памяти называется также корнем (root). То, что у нас получилось, схематически представлено на рис. 3.21.

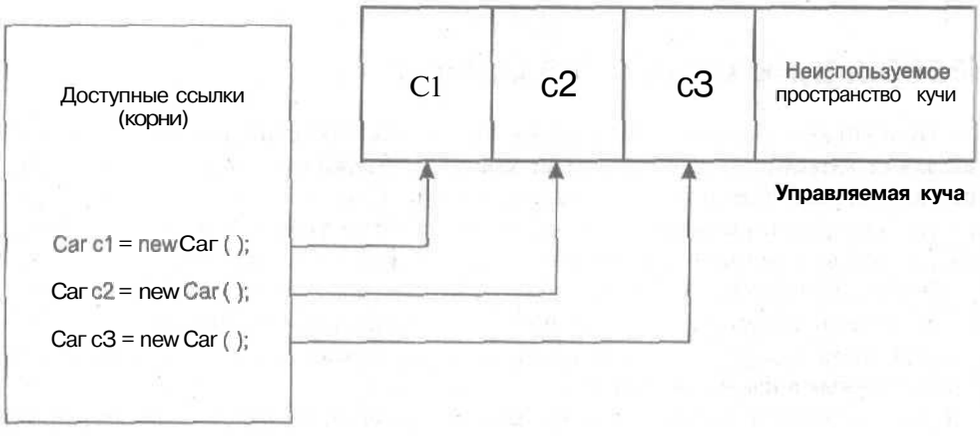


Рис. 3.21, Ссылки указывают на местонахождение объектов в управляемой куче

Если вы занимаетесь только тем, что создаете все новые и новые объекты, в конце концов пространство в управляемой куче закончится. В ситуации, когда свободного места в управляемой куче больше нет, а вы пытаетесь создать новый объект, будет сгенерировано исключение OutOfMemoryException. Поэтому, если вы хотите создать код приложения, совершенно исключив возможность возникновения ошибок, создавать объекты можно следующим образом (в реальных приложениях так обычно не поступают):

```
// Создаем объекты Car таким образом, чтобы отреагировать на возможную нехватку места
// в управляемой куче
public static int Main(string[] args)
{
...
      Car yetAnotherCar;
      try
      {
            yetAnotherCar = new Car();
      }
      catch(OutOfMemoryException e)
      {
            Console.WriteLine(e.Message);
            Console.WriteLine("Managed heap is FULL! Running GC...");
      }
...
      return 0;
}.
```

Вне зависимости от того, насколько осторожно вы будете создавать объекты, как только место в управляемой куче заканчивается, автоматически запускается сборщик мусора (garbage collector, GC). Сборщик мусора оценивает все объекты, размещенные в настоящий момент в управляемой куче, с точки зрения того, есть ли в области видимости приложения активные ссылки на них. Если активных ссылок на какой-либо объект больше нет или объект установлен в `null`, этот объект помечается для удаления, и в скором времени память, занимаемая подобными объектами, высвобождается.

Завершение ссылки на объект

Та схема управления памятью, которая была рассмотрена выше, обладает одной важной особенностью, которая имеет как положительные, так и отрицательные стороны: она работает полностью автоматически. С одной стороны, это упрощает процесс программирования. С другой — нас может не устраивать то, что процесс удаления объектов (закрытия соединения с базой данных, окна Windows и т. п.) будет происходить в соответствии с неизвестным нам алгоритмом. Например, если тип `Car` устанавливает в процессе выполнения соединение с удаленным компьютером, скорее всего, мы захотим, чтобы это соединение разрывалось в соответствии с установленными нами правилами.

Если вам нужно обеспечить возможность удаления объектов из оперативной памяти в соответствии с определенными вами правилами, первое, о чем вам необходимо позаботиться — о реализации в вашем классе метода `System.Object.Finalize()`. Заметим между прочим, что реализация этого метода по умолчанию (в базовом классе) ничего не дает. Однако, как это ни странно, в C# запрещено напрямую замещать метод `Object.Finalize()`. Более того, вы даже не сможете вызвать в вашем приложении этот метод напрямую! Если вы хотите, чтобы ваш пользовательский класс поддерживал метод `Finalize()`, вы должны использовать в определении этого класса метод, очень похожий на деструктор C++:

```
// Что-то очень знакомое
public class Car : Object
{
        // Деструктор C#?
        ~Car()
        {
                // Закрывайте все открытые объектом ресурсы!
                // Далее в C# будет автоматически вызван метод Base.Finalize()
        }
```

Г

Если у вас есть опыт работы с C++, то подобный синтаксис вам покажется знакомым. В C++ деструктор класса — это специальный метод класса, имя которого выглядит как имя класса, перед которым стоит символ тильды (-). В C++ гарантируется, что этот метод будет вызван, когда ссылка на объект выходит за пределы области видимости (для типов, размещенных в стеке) или к объекту применяется оператор delete (для объектов, размещенных в области динамической памяти).

При размещении объекта C# в управляемой куче при помощи оператора new среда выполнения автоматически определяет, поддерживает ли ваш объект метод Finalize() (представленный в C# с помощью «деструктороподобного» синтаксиса). Если этот метод поддерживается объектом, ссылка на этот объект помечается как «завершаемая» (finalizable). При этом в специальной внутренней очереди «завершения» (finalization queue) помещается указатель на данный объект. Когда сборщик мусора приходит к выводу, что наступило время удалять данный объект из оперативной памяти, он обращается к этому указателю и запускает деструктор C#, определенный для этого класса, прежде чем будет произведено физическое удаление объекта из памяти.

Завершение в подробностях

Предположим, что мы определили несколько классов автомобилей (Mini van, SportsCar, Jeep и наш класс Car). Предположим, что классы Minivan и SportsCar не поддерживают деструктор C#, а классы Jeep и Car — поддерживают. Поскольку классы Jeep и Car тем самым замещают метод Object.Finalize() (косвенно), очередь завершения будет содержать указатели на любой активный объект этих классов. То, что при этом происходит, схематически можно представить так, как показано на рис. 3.22.

Как можно догадаться, удаление из памяти классов, для которых определены деструкторы C#, занимает больше времени, чем удаление классов, для которых деструкторы не определены. В нашем примере для объектов c1 и c4 деструкторы не определены, и поэтому, когда сборщик мусора сочтет это необходимым, они будут просто немедленно удалены из памяти. При удалении c2 и c3 (объектов с деструкторами) будут производиться дополнительные вызовы, определенные нами в соответствующих деструкторах. Однако если нам нужно убедиться, что при удалении объекта будут гарантированно освобождаться все занятые им ресурсы и освобождаться они будут в установленном нами порядке, использования пользовательского деструктора C# не избежать.

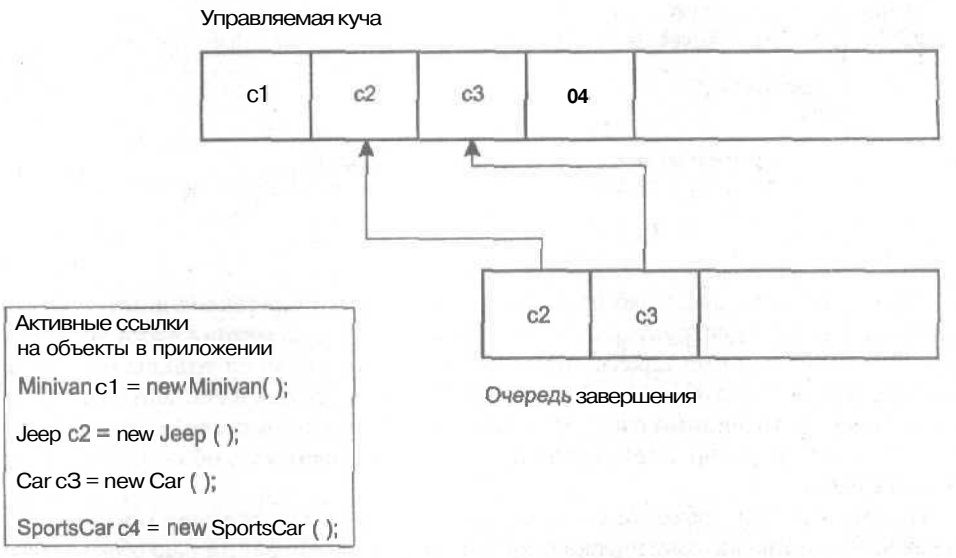


Рис. 3.22. Для объектов, поддерживающих деструктор C#, в очереди завершения создаются специальные указатели

Создание метода удаления для конкретного случая

Еще раз предположим, что объекты в нашем приложении используют определенные ресурсы. Конечно, все эти ресурсы освобождаются при помощи встроенных или пользовательских деструкторов (о которых было рассказано в предыдущем разделе). Однако существуют некоторые особо ценные ресурсы, например подключения к базам данных (за каждое подключение приходится платить), которые хотелось бы освобождать немедленно, а не в соответствии с расписанием работы сборщика мусора. Поэтому проблему можно сформулировать так: каким образом можно освободить ресурс, занятый объектом класса, немедленно, не дожидаясь естественной смерти этого объекта?

Наиболее очевидный способ — использовать для этого специальный метод. В C# метод, принудительно освобождающий интересующие пользователя ресурсы объекта, принято называть Dispose() (освободить). Этот метод пользователь объекта будет вызывать вручную, сразу же по завершении использования этого объекта и задолго до того, как объект выйдет за пределы области видимости и будет помечен как подлежащий физическому удалению из памяти (завершению). Таким образом, можно гарантировать освобождение ресурсов без помещения указателя на деструктор в очередь завершения. Кроме того, в этом случае освобождение ресурсов будет произведено немедленно, а не тогда, когда у сборщика мусора «дойдут руки» до нашего объекта:

```
// Снабжаем класс методом удаления для конкретного случая
public Car
{
```

```
    ...
    // Специальный метод, который пользователь объекта должен вызвать вручную
    public void Dispose()
    {
        // ...Закрываем открытые внутренние ресурсы
    }
}
```

Интерфейс IDisposable

Чтобы обеспечить единообразие методов, освобождающих ресурсы в разных классах, библиотеки классов .NET определяют интерфейс IDisposable, который содержит единственный член — метод Dispose():

```
public interface IDisposable
{
    public void DisposeO;
}
```

С концепцией интерфейсов мы познакомимся в следующей главе. Сейчас достаточно будет отметить, что рекомендуется реализовывать этот интерфейс для всех классов, которые должны поддерживать явную форму освобождения ресурсов. Например, применить этот интерфейс в случае класса Car можно следующим образом:

```
// Реализуем IDisposable
public Car : IDisposable
{
...
    // Это - единственный метод, который пользователь объекта должен вызвать
    // вручную
    public void DisposeO
    {
        // ...Закрываем открытые внутренние ресурсы
    }
}
```

Используя этот подход, мы предоставляем пользователю возможность в любой момент высвобождать наиболее ценные ресурсы, не загружая дополнительно очередь завершения. Кроме того, мы вполне можем сочетать применение интерфейса IDisposable (с методом Dispose()) и пользовательского деструктора C#.

Взаимодействие со сборщиком мусора

Как и все в мире .NET, сборщик мусора — это объект, и мы можем обращаться к нему через ссылку на объект. Для работы со сборщиком мусора в C# предназначен специальный класс — System.GC (от garbage collector — сборщик мусора). Этот класс определен как sealed, то есть производить от него другие классы при помощи наследования невозможно. В System.GC определен небольшой набор статических членов, при помощи которых и осуществляется взаимодействие со сборщиком мусора. Самые важные из этих членов представлены в табл. 3.2.

Мы проиллюстрируем взаимодействие со сборщиком мусора .NET на примере нашего любимого класса Car:

```
// Пример очистки памяти
public class Car : IDisposable
{
...
        ~Car()
        {
                // При сборке мусора вызвать наш собственный вариант метода Dispose()
                Dispose();
        }

        // Наш вариант метода Dispose()
        public void DisposeO
        {
                // Закрываем открытые внутренние ресурсы
                // Если пользователь вызвал Dispose(), необходимость в завершении
                // отпадает, поэтому подавляем завершение
                GC.SupressFinalize(this);
        }
}
```

Таблица 3.2. *Некоторые* члены типа System.GC

Член	Назначение
Collect()	Заставляет сборщик мусора заняться выполнением своих обязанностей для всех поколений (о поколениях — чуть ниже). По желанию можно указать в качестве параметра конкретное поколение
GetGeneration()	Возвращает поколение, к которому относится данный объект
MaxGeneration	Возвращает максимальное количество поколений, поддерживаемое данной системой
ReRegisterForFinalize()	Устанавливает флаг возможности завершения для объектов, которые ранее были помечены как незавершаемые при помощи метода SupressFinalize()
SuppressFinalize()	Устанавливает флаг запрещения завершения для объектов, которые в противном случае могли бы быть завершены сборщиком мусора
GetTotalMemory()	Возвращает количество памяти (в байтах), которое в настоящее время занимают объекты в управляемой куче, включая те объекты, которые будут вскоре удалены. Этот метод принимает параметр типа boolean, с помощью которого можно указать, запускать или нет процесс сборки мусора при вызове этого метода

Обратите внимание, что этот вариант класса Car поддерживает как деструктор C#, так и интерфейс IDisposable. Метод DisposeO определен таким образом, что при его выполнении происходит вызов метода GC.SupressFinalize(). Таким образом, мы сообщаем системе, что деструктор для данного объекта вызывать уже не нужно — все ресурсы будут освобождены при помощи метода DisposeO.

Мы покажем возможности сосуществования явного и подразумеваемого удаления объектов на примере программы, представленной ниже. В ней перед самым концом программы происходит вызов метода GC.CollectO, который инициирует срабатывание деструкторов для всех объектов. Однако ранее мы произвели вызов метода DisposeO для двух объектов Car. В результате для этих объектов сработал метод GC.SupressFinalize() (смотри определение класса Car выше), и эти объекты

были помечены как незавершаемые. Поэтому в процессе работы GC.Collect() деструкторы для этих двух объектов вызваны не будут:

```
// Пример взаимодействия с GC
public class GCApp
{
        public static int Main(string[] args)
        {
                Console.WriteLine("Heap memory in use: {0}",
                                        GC.GetTotalMemory(false).ToString());

                // Размещаем объекты класса Car в управляемой куче
                Car c1, c2. c3, c4;

                c1 = new Car("Car one". 40. 10);
                c2 - new Car("Car two", 70. 5):
                c3 - new Car("Car three", 200. 100);
                c4 = new Car("Car four", 140. 80):

                // Применяем метод Dispose() к некоторым объектам. В результате завершение
                // для них будет отменено
                c1.Dispose();
                c3.Dispose();

                // Вызываем метод Finalize() для объектов, остающихся в очереди
                // завершений
                GC.CollectO;

                return 0;
        }
}
```

Оптимизация сборки мусора

При знакомстве с членами класса System.GC мы уже встречались с понятием поколения. Поколение (generation) — это еще одна концепция, имеющая назначение сделать процесс сборки мусора в .NET более удобным.

Когда сборщик мусора .NET помечает объекты для завершения, обычно он не проверяет все подряд объекты приложения: это заняло бы слишком много времени, особенно для больших (то есть реальных) приложений. Для того чтобы повысить производительность сборки мусора, все объекты в управляемой куче разбиты на группы — поколения. Смысл такой группировки прост: чем дольше объект существует в управляемой куче, тем больше вероятность того, что он будет нужен и в дальнейшем. В качестве примера можно привести объект самого приложения — он появляется при запуске приложения и удаляется лишь при завершении его работы. В то же время существует значительная вероятность того, что недавно появившиеся объекты быстро перестанут быть нужными (например, временные объекты, определенные внутри области видимости метода). Основываясь на этой концепции, каждый объект относится к одному из следующих поколений:

- Поколение 0: недавно появившиеся объекты, которые еще не проверялись сборщиком мусора.

- Поколение 1: объекты, которые пережили одну проверку сборщика мусора (они были помечены для удаления, но не удалены физически, поскольку в управляемой куче было достаточно свободного места).
- Поколение 2: объекты, которые пережили более чем одну проверку сборщика мусора.

При очередном запуске процесса сборки мусора сборщик в первую очередь производит проверку и удаление всех объектов поколения 0. Если при этом освободилось достаточно места, выжившие объекты поколения 0 переводятся в поколение 1 и на этом процесс сборки мусора заканчивается. Если же после проверки поколения 0 места все еще недостаточно, запускается процесс проверки объектов поколения 1, а затем (при необходимости) — и поколения 2. Таким образом, за счет концепции поколений недавно созданные объекты обычно удаляются быстрее, чем объекты «с историей».

Мы вполне можем определять, к какому поколению относится тот или иной объект, непосредственно в процессе выполнения программы. Для этого используется метод GC.GetGeneration(). Кроме того, метод GC.Collect() позволяет нам указать поколение, которое будет проверяться при вызове сборщика мусора. Код программы, использующей эти возможности, может выглядеть следующим образом:

```
// Сколько уже прожил наш объект?
public static int Main(string[] args)
{
        Console.WriteLine("Heap memory in use: " + GC.GetTotalMemory(false).ToString());

        // Размещаем объекты класса Car в управляемой куче
        Car c1, c2, c3, c4;
        c1 = new Car("Car one", 40, 10);
        c2 = new Car("Car two", 70, 5);
        c3 = new Car("Car three", 200, 100);
        c4 = new CarC'Car four", 140, 80);

        // Выводим информацию о принадлежности объектов к поколениям
        Console.WriteLine("C1 is gen {0}", GC.GetGeneration(c1));
        Console.WriteLine("C2 is gen {0}", GC.GetGeneration(c2));
        Console.WriteLine("C3 is gen {0}", GC.GetGeneration(c3));
        Console.WriteLine("C4 is gen {0}", GC.GetGeneration(c4));

        // Удаляем вручную некоторые объекты
        c1.Dispose();
        c2.Dispose();

        // Запускаем процесс сборки мусора для объектов поколения 0
        GC.Collect(0);

        // Вновь выводим информацию о поколениях (все объекты перейдут в следующее
        // поколение
        Console.WriteLine("C1 is gen {0}", GC.GetGeneration(c1));
        Console.WriteLine("C2 is gen {0}", GC.GetGeneration(c2));
        Console.WriteLine("C3 is gen {0}", GC.GetGeneration(c3));
        Console.WriteLine("C4 is gen {0}", GC.GetGeneration(c4));

        // Запускаем процесс сборки мусора для всех поколений
```

```
        GC.Collect(); // Вызываем деструкторы для всех объектов, остающихся в куче

        Console.WriteLine("Heap memory in use: " + GC.GetTotalMemory(false).ToString());

        return 0;
    }
```

Результат выполнения этой программы представлен на рис. 3.23. Обратите внимание, что после вызова процесса сборки мусора для поколения 0 все объекты перешли в поколение 1 (поскольку они смогли пережить эту проверку).

Существует общее правило: обращаться к сборщику мусора следует только тогда, когда это действительно необходимо. Среда выполнения .NET спроектировала таким образом, чтобы процесс сборки мусора был полностью автоматизирован и программисту не пришлось задумываться о ручном удалении объектов в коде программы. Необдуманное вмешательство в процесс сборки мусора может привести к снижению производительности приложения. Однако в случае необходимости мы вполне можем определить логику нашего пользовательского деструктора C# или реализовать интерфейс IDisposable для принудительного освобождения ресурсов объектом нашего класса.

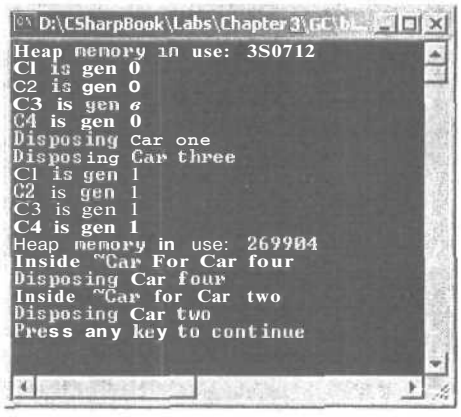


Рис. 3.23. Взаимодействие со сборщиком мусора

Подведение итогов

Если вы уже были знакомы с каким-либо объектно-ориентированным языком, в процессе чтения этой главы вы отметите только основные отличия C# от известного вам языка. Если же вы ранее не были связаны с объектно-ориентированным программированием, многие концепции этой главы могут оказаться для вас новыми. Однако в любом случае информация, представленная в этой главе, относится к основополагающим принципам любых приложений и классов .NET.

В начале этой главы были рассмотрены «три столпа» объектно-ориентированного программирования: инкапсуляция, наследование и полиморфизм. C# полностью поддерживает каждую из этих концепций. Кроме того, в этой главе была рас-

смотрена структурированная обработка исключений — основной механизм для работы с информацией об ошибках в .NET.

Эта глава завершилась рассмотрением вопросов, связанных с освобождением оперативной памяти средой выполнения .NET. Мы познакомились с методом `Object.Finalize()` и возможностями работы с ним в C#, с интерфейсом `IDisposable` и методом `Dispose()`, созданием деструкторов C# и средствами программного взаимодействия с объектом `System.GC`, представляющим сборщик мусора среды выполнения .NET.

Интерфейсы и коллекции 4

Эта глава посвящена применению интерфейсов в C#. Мы узнаем, как создаются и реализуются пользовательские интерфейсы и как создавать типы, поддерживающие множественное поведение. Кроме этого, мы также обсудим такие темы, как получение ссылок на интерфейсы, косвенную реализацию интерфейсов и создание иерархий интерфейсов.

В конце этой главы мы рассмотрим стандартные интерфейсы, определенные в библиотеках базовых классов .NET. Вы можете использовать эти интерфейсы в своих пользовательских классах для реализации поддержки таких возможностей, как копирование объектов, перечисление объектов и сортировка объектов. Мы также познакомимся с готовыми для использования в приложениях встроенными интерфейсами, реализованными в системных классах коллекций, таких как `ArrayList`, `Stack` и прочих. Эти классы определены в пространстве имен `System.Collections`.

Программирование с использованием интерфейсов

Пожалуй, самую важную роль концепция интерфейсов играет в COM-программировании. COM буквально построен на использовании интерфейсов: единственный способ, с помощью которого клиент COM может взаимодействовать с классом COM, основан на использовании указателя на интерфейс (а не ссылке на объект напрямую). В .NET использование интерфейсов — это уже не единственный способ реализации взаимодействия между двумя двоичными файлами (среда выполнения .NET поддерживает применение ссылок на объекты). Однако вряд ли кому-нибудь придет в голову, что интерфейсы в .NET больше не нужны. И в .NET эти синтаксические сущности — наиболее удобный способ расширить функциональность пользовательского типа без риска испортить существующий код.

Формальное определение интерфейса звучит так: интерфейс — это набор семантически связанных *абстрактных членов.* Количество членов, определенных

в конкретном интерфейсе, зависит от того, какое поведение мы пытаемся смоделировать при помощи этого интерфейса. С точки зрения синтаксиса интерфейсы в C# определяются следующим образом:

```
// Этот интерфейс определяет возможности "работы с углами" геометрической фигуры
public Interface IPointy
{
        byte GetNumberOfPoints();    // Автоматически (неявным образом) этот член
                                     // интерфейса становится абстрактным
}
```

Интерфейсы .NET также могут поддерживать любое количество свойств (и событий). Например, интерфейс IPointy может содержать такое свойство для чтения и записи:

```
public Interface IPointy
{
        // Чтобы сделать это свойство свойством "только для чтения" или "только
        // для записи", достаточно просто удалить соответствующий блок set или get
        byte Points{get; set;}
}
```

В любом случае интерфейс — это не более чем именованный набор абстрактных членов, а это значит, что любой класс, реализующий этот интерфейс, должен самостоятельно полностью определять каждый из членов этого интерфейса. Таким образом, интерфейсы — это еще один способ реализации полиморфизма в приложении: поскольку в разных классах члены одних и тех же интерфейсов будут реализованы по-разному, в результате эти классы будут реагировать на одни и те же вызовы по-своему.

Наш интерфейс IPointy — это элементарный интерфейс, определяющий поведение, связанное с углами (points) геометрических фигур. В качестве примера мы будем использовать уже знакомую нам по главе 3 иерархию геометрических фигур, производных от базового класса Shape. Главная идея проста: у некоторых геометрических фигур (например, у шестиугольника — Hexagon, и треугольника — Triangle) есть углы, и в них будет реализован этот интерфейс, а у других (окружность — Circle) углов нет, и соответствующие классы этот интерфейс поддерживать не будут. Если вы знаете, что какие-то классы поддерживают известный вам интерфейс, то вы вправе ожидать, что они будут должным образом реагировать на методы этого интерфейса. Например, если в Hexagon и Triangle реализован интерфейс IPointy, то эти классы должны ожидаемым образом реагировать на метод GetNumberOfPoints().

К этому моменту у вас мог созреть один интересный вопрос. Как мы уже выяснили, интерфейс — это набор абстрактных членов. Однако C# позволяет нам использовать абстрактные члены и в обычном классе. Так зачем же вообще нужны интерфейсы, если мы можем создать набор абстрактных методов, реализовав его как класс, и сделать этот класс базовым для наших пользовательских классов? Ответ будет прост: класс — это класс, а интерфейс — это интерфейс. В классах помимо абстрактных методов, свойств и событий определяются также переменные класса и обычные (не абстрактные) методы, появление которых в интерфейсе исключено.

Интерфейс — это чистая синтаксическая конструкция, которая предназначена только для определенных целей. Интерфейсы никогда не являются типами дан-

ных, и в них не бывает реализаций методов по умолчанию. Каждый член интерфейса (будь то свойство или метод) автоматически становится абстрактным. Кроме того, вспомним, что в C# (и во всем мире .NET) наследование одного класса более чем от одного базового класса (то есть множественное наследование) запрещено. В то же время реализация в классе сразу нескольких интерфейсов — это обычное дело.

Реализация интерфейса

Когда в C# какой-либо класс должен реализовать нужные нам интерфейсы, названия этих интерфейсов должны быть помещены (через запятую) после двоеточия, следующего за именем этого класса. Обратите внимание, что имя базового класса всегда должно стоять перед именами любых интерфейсов:

```
// Любой класс может реализовывать любое количество интерфейсов, но он должен
// производиться только от одного базового класса:
public class Hexagon : Shape, IPointy
{
        public Hexagon(){}
        public Hexagon(string name): base(name){}

        public override void Draw()
        {
                // Вспомним, что в классе Shape определено свойство PetName
                Console.WriteLine("Drawing {0} the Hexagon", PetName);
        }

        // Реализация IPointy
        public byte GetNumberOfPoints()
        {
                return 6;
        }
}

public class Triangle : Shape, IPointy
{
        public Triangle(){}
        public Triangle(string name) : base(name) {}

        public override void Draw()
        {
                Console.WriteLine("Drawing {0} the Triangle", PetName);
        }

        // Реализация IPointy
        public byte GetNumberOfPoints()
        {
                return 3;
        }
}
```

Теперь и Hexagon, и Triangle, когда их об этом попросят, предоставят информацию о том, сколько у них углов. Обратите внимание, что вы не можете выбирать, какие методы в интерфейсе вам реализовывать, а какие — нет. В классе либо должны быть реализованы все методы данного интерфейса, либо этот интерфейс вообще не реализуется — половинчатого решения быть не может.

Чтобы легче было представить настоящее состояние нашей иерархии геометрических фигур, обратимся к рис. 4.1. Программистам, работавшим с COM, будет, без сомнения, хорошо знаком такой «леденцовый» (lollipop) способ отображения иерархии классов и поддерживаемых этими классами интерфейсов. Для них же (программистов, пришедших из мира COM) отметим, что классы Hexagon и Triangle *не реализуют* интерфейс IUnknown и производятся от общего базового класса.

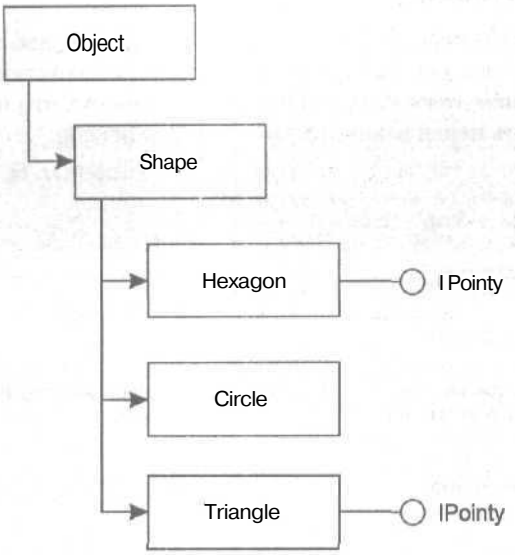


Рис. 4.1. Иерархия геометрических фигур с интерфейсами

Получение ссылки на интерфейс

C# позволяет обращаться к членам интерфейсов несколькими способами. Предположим, что мы создаем объект одного из наших классов и нам необходимо вызвать для него метод GetNumberOfPoints(), входящий в наш интерфейс IPointy.

Первый способ — воспользоваться явным приведением типов:

```
// Получаем ссылку на интерфейс IPointy. используя явное приведение типов
Hexagon hex = new Hexagon("Bill");
IPointy itfPt - (IPointy)hex;
Console.WriteLine(itfPt.GetNumberOfPoints());
```

Здесь мы получаем ссылку на интерфейс IPointy, явно приводя объект класса Hexagon к типу IPointy. Если класс Hexagon поддерживает интерфейс IPointy, мы получим ссылку на интерфейс (у нас она называется itfPt) и все будет хорошо. Однако что произойдет, если мы попробуем применить то же самое приведение типов к объекту класса, не поддерживающего IPointy (например, класса Circle)? Результат будет неутешителен: мы получим сообщение об ошибке времени выполнения (рис. 4.2). Когда мы пытаемся получить ссылку на интерфейс путем явного приведения типов для объекта класса, не поддерживающего данный интерфейс, система генерирует исключение InvalidCastException.

Чтобы избежать проблем с исключением, исключение нужно перехватить:

```
// Используя программные средства, поэтапно перехватываем исключение
Circle c - new Circle("Lisa");
IPointy itfPt;
try
{
        itfPt = (IPointy)c;
        Console.WriteLine(itfPt.GetNumberOfPoints());
}
catch(InvalidCastException e)
{ Console.WriteLine("OOPS! Not pointy..."); }
```

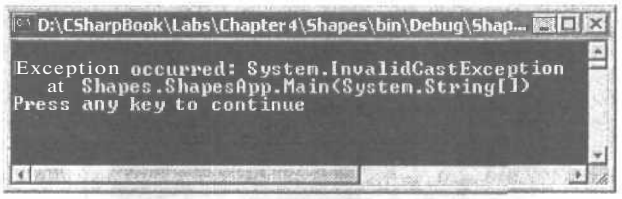


Рис. 4.2. Ошибка при приведении типов

Второй способ получить ссылку на интерфейс — использовать ключевое слово as:

```
// Еще один способ получить ссылку на интерфейс
Hexagon hex2 = new Hexagon("Peter");
IPointy itfPt2:
itfPt2 - hex2 as IPointy;
if(itfPt2 != null)
        Console.WriteLine(itfPt2.GetNumberOfPoints());
else
        Console.WriteLine("OOPS! Not pointy...");
```

Если при использовании ключевого слова as мы попробуем создать ссылку на интерфейс через объект, который этот интерфейс не поддерживает, ссылка будет просто установлена в null, и при этом никаких исключений генерироваться не будет.

Третий способ получения ссылки на интерфейс — воспользоваться оператором is. Если объект не поддерживает интерфейс, условие станет равно false:

```
// Есть ли у тебя углы?
Triangle t = new TriangleC);
if(t is IPointy)
        Console.WriteLine(t.GetNumberOfPoints());
else
        Console.WriteLine("OOPS! Not pointy...");
```

Если у нас имеется массив разных объектов и нам необходимо будет выяснить в процессе выполнения, какие именно объекты из этого массива поддерживают определенный интерфейс, это можно сделать любым из приведенных выше способов. Например, так:

```
// Давайте выясним (во время выполнения), у каких геометрических фигур есть углы
Shape[] s = {new Hexagon(), new Circle(), new Triangle("Joe"), new Circle("JoJO")}
```

```
for(int i = 0; i < s.length; i++)
{
        // Вспомним, что базовый класс Shape() определяет абстрактный метод Draw()
        s[i].Draw();

        // У каких геометрических фигур в массиве есть углы?
        if(s[1] is IPointy)
                Console.WriteLine("Points: {0}", ((IPointy)s[i]).GetNumberOfPoints());
        else
                Console.WriteLine(s[i].PetName + "\'s not pointy!");
}
```

Результат выполнения этой программы представлен на рис. 4.3.

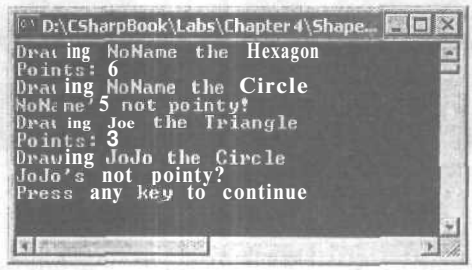


Рис. 4.3. Во время выполнения программы определяем, как себя будет вести объект

Интерфейсы как параметры

Интерфейсы можно рассматривать как сугубо специальные переменные. Это сходство подтверждается тем, что C# позволяет использовать интерфейсы как параметры, принимаемые и возвращаемые методами. Для наглядности представим, что мы определили еще один интерфейс с именем IDraw3D:

```
// Интерфейс для отображения фигур в трех измерениях:
public interface IDraw3D
{
        void Draw3D();
}
```

Предположим, что этот интерфейс реализован только в двух из трех наших классов для геометрических фигур — в классах Circle и Hexagon:

```
// Circle поддерживает интерфейс IDraw3D
public class Circle : Shape, IDraw3D
{
...
        public void Draw3D()
        {
                Console.WriteLine("Drawing Circle in 3D!");
        }
}

// Если наши типы поддерживают несколько интерфейсов, нужно просто перечислить
// эти интерфейсы через запятую, как обычно:
public class Hexagon : Shape, IPointy, IDraw3D
```

```
{
...
        public void Draw3D()
        {
                Console.WriteLine("Drawing Hexagon in 3D!");
        }
}
```

Если определить какой-нибудь метод, принимающий интерфейс IDraw3D в качестве параметра, можно будет просто передавать этому методу любой объект, поддерживающий IDraw3D, например:

```
// Создаем несколько геометрических фигур. Если они поддерживают отображение
// в трех измерениях, делаем это!
public class ShapesApp
{
        // Будут нарисованы все объекты, поддерживающие интерфейс IDraw3D
        public static void DrawThisShapeIn3D(IDraw3D itf3d)
        {
                itf3d.Draw3D();
        }

        public static int Main(string[] args)
        {
                Shape[] s = {new Hexagon(), new Circle(),
                             new Triangle(), new Circle("JoJo")};

                for(int i=0; i<s.Length; i++)
                {
                        // Могу ли я нарисовать этот объект в трех измерениях?
                        if(s[i] is IDraw3D)
                                DrawThisShapeIn3D((IDraw3D)s[i]);
                }
                return 0;
        }
}
```

Результат работы этой программы представлен на рис. 4.4. Поскольку Triangle (треугольник) у нас не поддерживает интерфейс IDraw3D, то он и не будет нарисован.

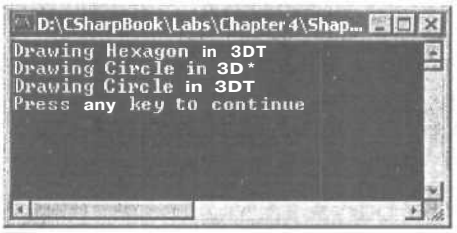


Рис. 4.4. Выясняем, какие типы поддерживают интерфейс IDraw3D

Явная реализация интерфейса

В нашем предыдущем примере мы назвали единственный метод интерфейса IDraw3D именем Draw3D() — главным образом, чтобы избежать конфликта имен с методом Draw(), определенным в базовом классе Shape:

```
// Интерфейс для отображения геометрических фигур в трех измерениях
public interface IDraw3D
{
        void Draw3D();
}
```

В принципе, такое определение интерфейса и метода вполне нас может устроить. Однако более удобным именем метода, пожалуй, все-таки было имя Draw():

```
// Интерфейс для отображения геометрических фигур в трех измерениях
public interface IDraw3D
{
        void Draw();
}
```

Однако не возникнет ли у нас проблем, если мы попытаемся создать класс, одновременно производный от класса Shape() и реализующий интерфейс IDraw3D? Ведь теперь метод с именем Draw() и унаследован от базового класса, и получен от интерфейса IDraw3D.

```
// Вопросы...
public class Line : Shape, IDraw3D // И базовый класс, и интерфейс определяют метод
                                   DrawO
{
        public override void DrawO
        {
                Console.WriteLine("Drawing a line...");
        }
}
```

Компилятор, как несложно убедиться, вовсе не против такого кода. Однако что произойдет, если мы обратимся к объекту класса L1 пе таким образом:

```
// Вызываем Line.Draw()
Line myLine = new Line();
myLine.Draw();

// Вызываем Line.Draw() еще раз. но уже по-другому
Idraw3D itfDraw3d - (IDraw3D) myLine;
itfDraw3d.Draw();
```

Исходя из того что мы знаем о базовом классе Shape и интерфейсе IDraw3D, очень похоже на то, что наследуют сразу два абстрактных метода DrawO. Для обоих абстрактных методов класс Line предлагает единую конкретную реализацию DrawO, и это вполне допустимо. Как мы бы ни вызвали потом этот метод — через ссылку на объект класса или через ссылку на интерфейс, — все равно будет вызван тот же самый вариант метода.

Однако проблемы еще не кончились. Что, если нам захочется иметь два метода с одинаковыми именами — IDraw3D.Draw() для отображения объекта в трех измерениях и замещенный метод Shape.Draw() для обычного его представления на плоскости? Вопрос можно и немного переформулировать. Как сделать так, чтобы к методам интерфейса можно было обратиться только через ссылку на интерфейс (но не через ссылку на объект)? В нашей ситуации к методу DrawO можно обратиться обоими способами.

Ответ на эти вопросы состоит в том, чтобы воспользоваться явной реализацией интерфейса (explicit interface implementation). Этим мы решаем сразу обе пробле-

мы — и устраняем потенциальный конфликт имен, и запрещаем обращаться к методам интерфейса иначе как через ссылку на интерфейс. В нашей ситуации явная реализация интерфейса может выглядеть следующим образом:

```
// При помощи явной реализации методов интерфейса мы можем определить
// разные варианты метода Draw()
public class Line : Shape. IDraw3D
{
        // Этот метод можно будет вызвать только через ссылку на интерфейс IDraw3D
        void IDraw3D.Draw()
        {
                Console.WriteLine("Drawing a 3D line...");
        }
        // Этот метод можно будет вызвать только через ссылку на объект класса Line
        public override void DrawO
        {
                Console.WriteLine("Drawing a line...");
        }
}
```

При использовании явной реализации интерфейса необходимо помнить о некоторых ее тонкостях. Во-первых, мы уже не сможем задать модификатор области видимости для методов интерфейса:

```
// Стоп! Так нельзя.
public class Line : Shape. IDraw3D
{
        public void IDraw3D.Draw()
        {
                Console.WriteLine("Drawing a 3D line...");
        }
...
}
```

Немного поразмыслив, можно понять, откуда взялся такой запрет. Как мы уже выяснили, единственное, для чего используется явная реализация интерфейса — так это для привязки методов интерфейса только на его уровне (чтобы к этим методам можно было обратиться только через ссылку на интерфейс). Если же мы используем модификатор области видимости public, то это будет значить, что мы перевели этот метод в группу открытых методов класса, а это путает нам все карты.

Во-вторых, использование явного объявления интерфейса — пожалуй, единственный способ уберечься от потенциальных конфликтов между именами методов разных интерфейсов. Например, предположим, что у вас имеется класс, который реализует все три указанных ниже интерфейса:

```
// Три интерфейса определяют методы с одинаковыми именами
public interface IDraw
{
        void DrawO;
}

public interface IDraw3D
{
        void DrawO;
}
```

```
public interface IDrawToPrinter
{
        void Draw();
}
```

Если вы хотите создать геометрическую фигуру, поддерживающую все три метода — Draw() для вывода обычного плоского изображения, Draw() для трехмерного изображения и Draw() для вывода изображения на принтер, единственный выход — воспользоваться явной реализацией интерфейса:

```
// Конфликтов имен не будет!
public class SuperImage : IDraw, IDrawToPrinter, IDraw3D
{
        void IDraw.Draw()
        {
                // Вывод обычного плоского изображения
        }
        void IDrawToPrinter.Draw()
        {
                // Вывод на принтер
        }
        void IDraw3D.Draw()
        {
                // Поддержка объемного изображения
        }
}
```

Код приложения Shapes можно найти в подкаталоге Chapter 4.

Создание иерархий интерфейсов

Как мы помним, в C# два класса могут вступать между собой в отношения наследования, при которых один класс станет базовым, а другой — производным. Точно так же один интерфейс C# может наследовать другому. Как обычно, базовый интерфейс (интерфейс более высокого уровня в иерархии) определяет общее поведение, в то время как производный интерфейс — более конкретное и специфическое. Простая иерархия интерфейсов может выглядеть следующим образом:

```
// Базовый интерфейс
interface IDraw
{
        void DrawO:
}

interface IDraw2 : IDraw
{
        void DrawToPrinter();
}

interface IDraw2 : IDraw3
{
        void DrawToMetaFile();
}
```

Отношения между этими интерфейсами представлены на рис. 4.5.

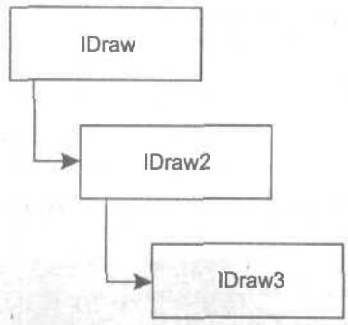


Рис. 4.5. Простая иерархия интерфейсов

Если наш класс должен поддерживать поведение, определенное во всех трех интерфейсах, он должен производиться от интерфейса самого нижнего уровня (в нашем случае — IDraw3). Все методы, определенные в базовых интерфейсах, будут автоматически включены в производные интерфейсы. Например:

```
// Этот класс будет поддерживать IDraw. IDraw2 и IDraw3
public class SuperImage : IDraw3
{
        // Используем явную реализацию интерфейсов, чтобы привязать методы
        // к конкретным интерфейсам
        void IDraw.Draw()
        {
                // Обычный вывод на экран
        }

        void IDraw2.DrawToPrinter()
        {
                // Вывод на принтер
        }

        void IDraw3.DrawToMetafile()
        {
                // Вывод в метафайл
        }
}
```

Применить этот класс на практике можно следующим образом:

```
// Проверяем наши интерфейсы
public class TheApp
{
        public static int Main(string[] args)
        {
                SuperImage si = new SuperImage();

                // Получаем ссылку на интерфейс IDraw
                IDraw itfDraw = (IDraw)si;
                itfDraw. DrawO;

                // А теперь получаем ссылку на интерфейс IDraw3
                if(itfDraw is IDraw3)
                {
                        IDraw3 itfDraw3 - CIDraw3)itfDraw;
```

```
                itfDraw3.DrawToMetaFile();
                itfDraw3.DrawToPrinter();
            }
            return 0;
        }
}
```

Результат работы нашей программы представлен на рис. 4.6.

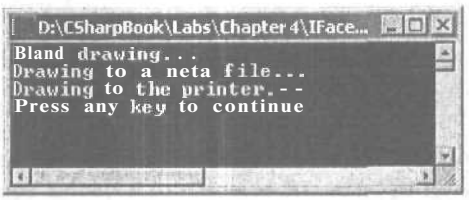


Рис. 4.6. Используем иерархию интерфейсов в SuperImage

Наследование от нескольких базовых интерфейсов

C#, в отличие от классического COM, вполне допускает наследование сразу от нескольких базовых интерфейсов. При этом еще раз заметим, что множественное наследование между классами (когда один класс является производным одновременно от нескольких классов) в C# запрещено — множественное наследование разрешается только для интерфейсов. Проиллюстрируем возможности множественного наследования интерфейсов на примере.

Предположим, что в нашем распоряжении есть набор интерфейсов, моделирующих поведение автомобиля:

```
interface IBasicCar
{
    void DriveO:
}
interface IUnderwaterCar
{
    void Dive();
}
// А это - интерфейс, производный сразу от двух предыдущих:
interface IJamesBondCar: IBasicCar, IUnderWaterCar
(
    void TurboBoost();
}
```

Если мы захотим создать класс, который реализует интерфейс IJamesBondCar, нам придется реализовать в нем все методы каждого из интерфейсов. В нашей ситуации таких методов три: TurboBoostO — разгоняться, DiveO — нырять и DriveO - ехать:

```
public class JBcar ; IJamesBondCar
{
    public JBCar(){}

    // Унаследованные члены
    void IBasicCar.Drive(){Console.WriteLine("Speeding up...");}
```

```
        void IUnderwaterCar.Dive(){Console.WriteLine ("Submerging...");}
        vo.id IJamesBondCar.TuboBoost(){Console.WriteLine("Blast off!");}
}
```

Теперь наш универсальный автомобиль поддерживает все три способа передвижения:

```
JBCar j = new JBCar();
if(j is IJamesBondCar)
{
        ((IJamesBondCar)j).Drive();
        ((IJamesBondCar)j).TurboBoost();
        ((IJamesBondCar)j).Dive();
}
```

Код приложения IfaceHierarchy можно найти в подкаталоге Chapter 4.

Создание пользовательского нумератора (интерфейсы IEnumerable и IEnumerator)

К этому моменту мы уже рассмотрели работу пользовательских интерфейсов и их иерархий. Настало время познакомиться с интерфейсами, которые уже встроены в библиотеку базовых классов .NET, и научиться приемам работы с ними. Стоит углубиться в библиотеку базовых классов .NET, как обнаружится, что множество стандартных классов-полуфабрикатов .NET реализуют одни и те же стандартные интерфейсы. Конечно, мы можем создавать и свои пользовательские типы, поддерживающие те же самые интерфейсы. То, как это делается, мы рассмотрим на примере интерфейсов IEnumerable и IEnumerator.

Для этого примера нам потребуется класс Cars, который представляет собой набор уже знакомых нам объектов класса Car (определенных в главе 3). Вот исходное определение класса Cars:

```
// Cars - набор объектов класса Car
public class Cars
{
        private Car[] carArray;

        // При создании объекта класса Cars заполняем его несколькими объектами Car
        public Cars()
        {
                carArray = new Car[4];
                carArray[0] - new Car("FeeFee", 200, 0);
                carArray[1] = new Car("Clunker", 90, 0);
                carArray[2] - new Car("Zippy", 30, 0);
                carArray[I] - new Car("Fred", 30, 0);
        }
}
```

С точки зрения применения класса Cars в приложении было бы очень удобно обращаться ко всем внутренним объектам Car этого класса при помощи конструкции foreach:

```
// Кажется очень заманчивым
public class CarDriver
{
```

```
        public static void Main()
        {
                Cars carLot = new Cars();
                // Пробуем использовать foreach для обращения к каждому объекту Car внутри
                // набора, представленного carLot
                "foreach(Car c in carLot){"
                {
                        Console.WriteLine("Name: {0}", c.PetName);
                        Console.WriteLine("Max speed: {0}", c.MaxSpeed);
                }
        }
}
```

Однако если вы попробуете запустить этот код на выполнение, компилятор начнет возмущаться тем, что класс Cars не реализует метод GetEnumerator(). Этот метод определяется в интерфейсе IEnumerable, который находится в пространстве имен System.Collections. Чтобы справиться с возникшей проблемой, необходимо переопределить класс Cars таким образом, чтобы он поддерживал и этот интерфейс, и этот метод:

```
// Для применения конструкции foreach необходимо, чтобы класс реализовывал интерфейс
IEnumerable
public class Cars : IEnumerable
{
...
        // Интерфейс IEnumerable определяет этот метод (и ничего больше):
        public IEnumerator GetEnumerator()
        {
                /7 А дальше-то что?
        }
```

Г

Если посмотреть, что делает GetEnumeratorO, то выяснится, что он возвращает еще один интерфейс — IEnumerator. Этот интерфейс используется для обращения к членам внутреннего набора объектов. Он также находится в пространстве имен System.Collections и определяет следующие три метода:

```
// GetEnumeratorO возвращает что-нибудь из нижеперечисленного
public interface IEnumerator
{
        bool MoveNext();            // Передвинуть внутренний указатель (курсор)
                                    // на одну позицию
        object Current {get;}       // Получить текущий элемент набора (свойство только
                                    // для чтения)
        void Reset();               // Установить курсор в начало набора (на первый
                                    // объект)
}
```

Учитывая, что IEnumerable.GetEnumerator() возвращает интерфейс IEnumerator, класс Cars придется переделать следующим образом:

```
// Уже теплее
public class Cars: IEnumerable. IEnumerator
{
...
        // Реализация IEnumerable
        public IEnumerator GetEnumeratorO
```

```
    {
        return (IEnumerator)this;
    }
```

Г

Последнее, что мы должны сделать, — наполнить реальным содержанием методы MoveNextO, Current и Reset. Таким образом, окончательный вариант класса Car, поддерживающего IEnumerable и I Enumerator, может выглядеть так:

```
// Набор объектов Car с реализованным нумератором!
public class Cars: IEnumerator, IEnumerable
{
    private car[] carArray;

    // Переменная для текущей позиции элемента в массиве
    int pos = -1;

    public Cars 0
    { // Здесь мы создаем несколько объектов класса Car и добавляем их в массив  }

    // Реализация методов интерфейса IEnumerator
    public bool  MoveNext0
    {
        if(pos < carArray.Length)
        {
            POS++;
            return true:
        }
        else
            return false;
    }

    public void Reset() { pos - 0: }

    public object Current
    {
        get { return carArray[pos]; }
    }

    // Реализация метода интерфейса IEnumerable
    public IEnumerator GetEnumerator()
    {
        return (IEnumerator)this;
    }
}
```

Теперь осталось разобраться с тем, что мы получили, снабдив наш класс Cars поддержкой интерфейсов IEnumerator и lEnumerable.

Во-первых, теперь с нашим классом стало возможно работать при помощи синтаксиса foreach:

```
// Ура!
foreach (Car c in carLot)
t
    Console.WriteLine("Name: {0}", c.PetName);
    Console.WriteLine("Max speed: {0{", c.MaxSpeed);
}
```

Во-вторых, теперь в нашем распоряжении есть новые способы обращения к объектам класса Car, находящимся внутри объекта класса Cars (что очень похоже на интерфейс IEnumXXXX в COM):

```
// Обращаемся к объектам Car через IEnumerator
IEnumerator itfEnum;
itfEnum = (IEnumerator)carLot;

// Устанавливаем курсор на начало
itfEnum.Reset();

// Перемещаем курсор вперед на один шаг
itfEnum.MoveNext();

// Выбираем одну машину и включаем в ней радио
object curCar = itfEnum.Current;
((Car)curCar).CrankTunes(true);
```

Код приложения ObjEnum можно найти в подкаталоге Chapter 4.

Создание клонируемых объектов (интерфейс ICloneable)

В главе 2 рассказывалось о том, что в классе System.Object определен метод MemberwiseClone(). Этот метод используется для специального типа копирования объекта — когда реальное копирование не происходит, а вместо этого создается еще одна ссылка на область оперативной памяти, занимаемую данным объектом. Для такого типа копирования используется специальный термин — «поверхностное копирование» (shallow copy). Пользователи объектов не вызывают метод MemberwiseClone() напрямую — он вызывается автоматически, когда к объектам ссылочного типа применяется оператор назначения (=), то есть одна ссылка начинает указывать на ту же область оперативной памяти, что и другая.

Предположим, что в нашем распоряжении имеется класс Point (точка):

```
// Наш класс - это просто точка с координатами на плоскости
public class Point
{
        // Поля (открытые переменные)
        public int x, y;

        // Конструкторы
        public Point() {}
        public Point(int x, int y){this.x = x; this.y = y;}

        // Замещаем Object.ToString()
        public override string ToString()
        {return "X: " + x + " Y: " + y; }
}
```

Point — это класс, а следовательно, он относится к ссылочным типам (как мы помним из обсуждения ссылочных и структурных типов в главе 2). Если мы применим к нему оператор назначения (=), то есть метод MemberwiseClone(), настоящей копии объекта создано не будет — вместо этого появится еще одна ссылка на об-

ласть, занимаемую объектом в оперативной памяти. Но очень часто бывает нужно создавать настоящие, действительно отдельные копии объекта (deep copy — глубокое копирование). Для того чтобы можно было применять глубокое копирование к объектам нашего класса при помощи стандартных методов, наш класс должен реализовывать интерфейс ICIoneable.

В интерфейсе ICIoneable предусмотрен один-единственный метод — CloneO. Реализация этого метода, конечно же, зависит от того, какие внутренние данные определены в вашем классе. Однако смысл работы этого метода для всех классов будет одним и тем же: будет создан новый объект, и всем переменным этого объекта-копии будут присвоены значения соответствующих переменных исходного объекта. Давайте научим наш объект Point клонироваться:

```
// Реализуем в классе Point поддержку глубокого копирования
// через интерфейс ICloneable
public class Point : ICloneable
{
        // Данные о состоянии объекта
        public int x. y;

        // Конструкторы
        public Point() {}
        public Point(int x, int y){this.x - x: this.y = y;}

        // Реализуем единственный метод ICloneable
        public object CloneO
        {
              return new Point(this.x, this.y)
        }

        public override string ToString()
        {return "X: " + x + " Y: " + y: }
}
```

Теперь мы можем создавать полностью независимые от исходного объекта копии объекта Point. Выглядеть это может, к примеру, так:

```
// Обратите внимание, что CloneO возвращает "объект вообще". Чтобы получить из него
// нужный нам производный тип, придется провести явное преобразование типов
Point p3 = new Point(100. 100);
Point p4 = (Point)p3.Clone();

// Меняем p4.x (при этом p3.x не изменится)
p4.x = 0;

!! Проверяем, так ли это:
Console.WriteLine("Deep copying using Clone()");
Console.WriteLine(p3);
Console.WriteLine(p4);
```

Если этот код покажется вам слегка знакомым, так оно и должно быть. Когда в главе 2 мы разбирали особенности работы с ссылочными и структурными типами, мы выяснили, что для структурных типов (например, структур C# или int) всегда используется глубокое (побитовое) копирование. Однако если мы хотим обеспечить возможность использовать глубокое копирование для объекта ссылочного типа (класса), то лучше всего сделать это, реализовав интерфейс ICloneable.

Однако будем внимательны! Если наш класс является производным от одного из многочисленных стандартных встроенных классов-«полуфабрикатов» из библиотеки базовых классов C#, то вполне возможно, что этот интерфейс в нем уже реализован. Мы советуем перед тем, как приступать к собственной реализации ICloneable, выяснить этот вопрос, обратившись к электронной документации Visual Studio.NET.

Код приложения ObjClone можно найти в подкаталоге Chapter 4.

Создание сравниваемых объектов (интерфейс IComparable)

Еще один распространенный интерфейс IComparable, также определенный в пространстве имен System, позволяет производить сортировку объектов, основываясь на специально определенном внутреннем ключе. Формальное определение этого интерфейса выглядит следующим образом:

```
// Этот интерфейс позволяет определять место объекта среди других аналогичных
// объектов
interface IComparable
{
        int CompareTo(object o);
}
```

Давайте предположим, что наш класс Car уже снова изменен таким образом, чтобы поддерживать и прозвище (petName), и внутренний идентификатор (CarID). Пусть пользователь создал массив объектов Car следующим образом:

```
// Создаем массив объектов Car
Car[] myAutos = new Car[5];
myAutos[0] = new Car(123, "Rusty");
myAutos[1] = new Car(6, "Mary");
myAutos[2] = new Car(83, "Viper");
myAutos[3] = new Car(13, "NoName");
myAutos[4] = new Car(9873, "Chucky");
```

Как уже говорилось, в классе System.Array определен статический метод Sort(). С помощью этого метода можно упорядочить (по возрастанию) массив из элементов некоторых встроенных типов данных (например, таких как int и short). Однако что произойдет, если вызвать этот метод для массива объектов Car?

```
// Попробуем рассортировать автомобили?
Array.Sort(myAutos); // Что-то не выходит...
```

Если попробовать запустить этот код на выполнение, будет сгенерировано исключение ArgumentException со следующим комментарием: At least one object must implement IComparable («По крайней мере в одном объекте должен быть реализован IComparable»). Таким образом, чтобы можно было стандартным способом производить сортировку ваших пользовательских объектов, они должны реализовывать интерфейс IComparable. Поскольку этот интерфейс состоит из единственного метода CompareTo(), вся соль заключается в том, как будет реализован этот метод. Видимо, наиболее важное решение, которое мы должны принять, — это определить, по значению какой внутренней переменной будет производиться сортировка. Для

нашего типа Car самая подходящая переменная — это идентификатор автомобиля CarID:

```
// Такая реализация метода CompareTo() позволит сортировать объекты автомобилей
// по значению идентификатора - CarJD
public class Car : IComparable
{
...
        II Реализация IComparable
        int IComparable.CompareTo(object o)
        {
                Car temp = Car(o);
                if(this.CarID > temp.CarID)
                        return 1:
                if(this.CarID < temp.CarID)
                        return -1;
                else
                        return 0;
        }
}
```

Как видно из этого кода, все, что делает метод СотрагеТоО, состоит в том, что он сравнивает значение CarID для текущего объекта (того, для которого вызван этот метод) со значением CarlD для принимаемого объекта (того объекта, который передан этому методу в качестве входящего параметра). В зависимости от результатов сравнения выдается одно из трех возможных значений. Что каждое из этих значений может значить для тех, кто пользуется методом CompareTo(), показано в табл. 4.1.

Таблица 4.1. Значения, возвращаемые методом СотрагеТоО

Значение	Что оно значит
Любое число меньше нуля	Значение идентификатора у текущего объекта меньше, чем у принимаемого в качестве параметра
Нуль	Значения идентификаторов у текущего и принимаемого объекта равны
Любое число больше нуля	Значения идентификатора у текущего объекта больше, чем у принимаемого

Теперь, когда интерфейс IComparable и метод СотрагеТоО делают то, что от них требуется, мы можем спокойно производить сортировку объектов Car при помощи стандартного метода Sort():

```
// Применяем реализованный нами интерфейс IComparable на практике
public class CarApp
{
        public static int Main(string[] args)
        {
                // Создаем массив объектов Car
                Car[j myAutos - new Car[5];
                myAutos[0] - new Car(123, "Rusty");
                myAutos[1] = new Car(6. "Mary");
                myAutos[2] = new Car(83. "Viper");
                myAutos[3] = new Car(13. "NoName");
                myAutos[4] = new Car(9873. "Chucky");
```

```
            // Выводим информацию об автомобилях из неупорядоченного массива
            // на системную консоль
            Console.WriteLine("Here is the unordered set of cars:");
            foreach(Car c in MyAutos)
                  Console.WriteLine(c.ID + " " + c.PetName);

            // А теперь используем возможности только что реализованного
            // IComparable
            Array.Sort(myAutos);

            // Выводим информацию уже из упорядоченного массива
            Console.WriteLine("Here is the ordered set of cars:");
            foreach(Car c in myAutos)
                  Console.WriteLine(c.ID + " " c.PetName);

            return 0;
      }
}
```

То, что должно получиться при запуске этой программы, представлено на рис. 4.7.

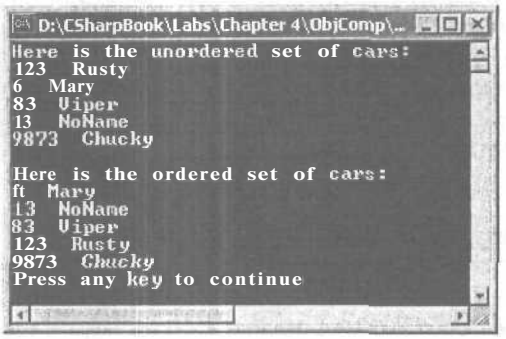


Рис. 4.7. Сортируем типы при помощи числового идентификатора

Если у нескольких объектов в массиве будет одинаковое значение идентификатора, по которому производится сортировка, то они будут расставлены в том же порядке относительно друг друга, в котором они поступили на операцию сортировки. Например, на рис. 4.8 у нас произошла именно такая ситуация: одинаковый ID встретился одновременно у трех объектов.

Сортировка по нескольким идентификаторам (IComparer)

В нашем предыдущем примере мы реализовали метод `CompareTo()` таким образом, чтобы сортировка объектов Car производилась по числовым значениям CarID. Совсем несложно чуть-чуть скорректировать реализацию этого метода, чтобы сортировка производилась по алфавитным значениям прозвища — PetName. Однако что делать, если нам нужно обеспечить возможность производить сортировку и по CarID, и по PetName — в зависимости от желания пользователя? Лучший выход из этой ситуации — воспользоваться возможностями еще одного стандартного интерфейса — IComparer, определенного в пространстве имен System.Collections.

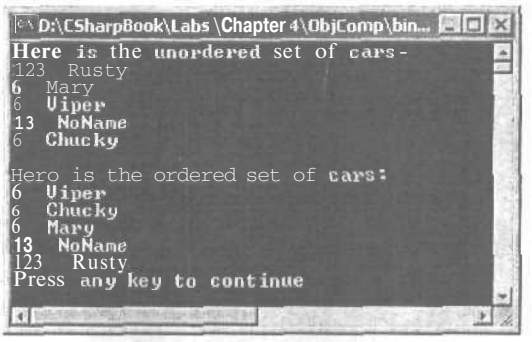


Рис. 4.8. Если у нескольких объектов идентификатор, по которому производится сортировка, одинаковый, они будут перечислены в порядке их подачи на сортировку

```
// Стандартный способ сравнения двух объектов
interface IComparer
{
        int Compare(object o1, object 02);
}
```

В отличие от интерфейса IComparable, IComparer обычно не реализуется напрямую внутри класса, объекты которого вы будете сортировать (в нашем случае — Car). Вместо этого этот интерфейс реализуется во вспомогательных классах, которых может быть любое количество — по одному вспомогательному классу на каждую переменную, по которой производится сортировка. Пока сортировка объектов Car может производиться только по идентификатору CarID. Чтобы класс Car одновременно мог поддерживать и сортировку по прозвищу — PetName, нам потребуется вспомогательный класс, реализующий IComparer. Выглядеть он может так:

```
// Этот вспомогательный класс нужен для сортировки объектов Car по PetName
using System.Collections;
public class SortByPetName : IComparer
{
        public SortByPetName(){}

        // Сравниваем прозвища (PetName) объектов
        int IComparer.Compare(object o1, object o2)
        {
                Car t1 = (Car)o1;
                Car t2 = (Car)o2;
                return String.Compare(t1.PetName, t2.PetName);
        }
}
```

Использовать этот вспомогательный класс очень просто: в System.Array предусмотрено множество вариантов перегруженного метода Sort(). Один из них как раз и предназначен для нашего случая, то есть для приема объекта, реализующего IComparer:

```
// Теперь мы можем сортировать Car еще и по прозвищу
Array.Sort(myAutos, new SortByPetName());

// Выводим информацию из упорядоченного массива на системную консоль
```

```
Console.WriteLine("Ordering by pet name:");
foreach(Car c in myAutos)
        Console.WriteLine(c.ID + " " + c.PetName);
```

То, что должно получиться, показано на рис. 4.9.

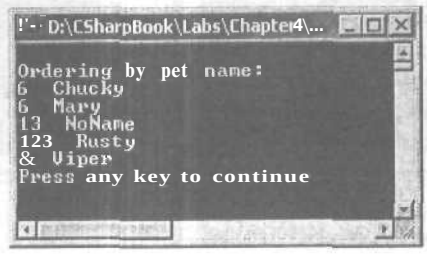


Рис. 4.9. Сортируем машины по прозвищу — в алфавитном порядке

Как с помощью специальных свойств сделать сортировку более удобной

В вызове того варианта перегруженного метода Array.Sort(), которым мы воспользовались выше, нет ничего сложного. Однако иногда пользователям класса бывает нелегко осознать, что для сортировки объектов, например, класса Car(), необходимо вначале рассмотреть совсем другой класс SortByPetName и понять, для чего он нужен. Если бы для передачи методу Array.Sort() использовался какой-либо член класса Car() с подходящим названием, было бы проще. Array.Sort() предоставляет нам такую возможность — существует еще один перегруженный вариант этого метода, который принимает в качестве одного из параметров статическое свойство только для чтения того класса, объекты которого мы сортируем. Но вначале нам нужно добавить такое свойство в определение класса:

```
// Добавляем в класс Car специальное свойство, которое будет возвращать интерфейс
IComparer
public class Car : IComparable
{
        ...
        // То самое свойство
        public static IComparer SortByPetName
        { get { return (IComparer)new SortByPetName(); } }

        ...
}
```

Такое свойство пользователь теперь вряд ли упустит из виду:

```
// Так можно, но лучше рассчитывать на всяких пользователей:
//Array.Sort(myAutos,new SortByPetName() );

// А вот так гораздо проще и естественней:
Array.Sort(MyAutos, Car.SortByPetName);
```

Код приложения ObjComp можно найти в подкаталоге Chapter 4.

Пространство имен System.Collections

Наиболее простой вариант набора элементов в C# — это массив System.Array. Однако он, как было сказано в главе 2, уже обладает весьма полезными встроенными функциями, которые позволяют производить операции сортировки, клонирования, перечисления и расстановки элементов в обратном порядке. Однако создатели библиотеки базовых классов C# приготовили для нас большое количество встроенных типов, которые позволят сэкономить массу времени при решении часто встречающихся задач. В этом разделе мы познакомимся со встроенными типами, определенными в пространстве имен System.Collections. Все эти типы, как следует из самого названия System.Collections, предназначены для работы с наборами элементов.

Первое, о чем необходимо сказать, — в System.Collections определен набор стандартных интерфейсов (многие из них нам уже пришлось реализовывать в примерах этой главы). Кроме того, эти же интерфейсы определены в большинстве классов System.Collections. Краткий перечень наиболее важных интерфейсов пространства имен System.Collections представлен в табл. 4.2.

Многие из этих интерфейсов объединены в иерархии, в то время как некоторые существуют независимо от остальных. Отношения наследования между интерфейсами представлены на рис. 4.10 (как мы помним, для интерфейсов вполне допускается множественное наследование).

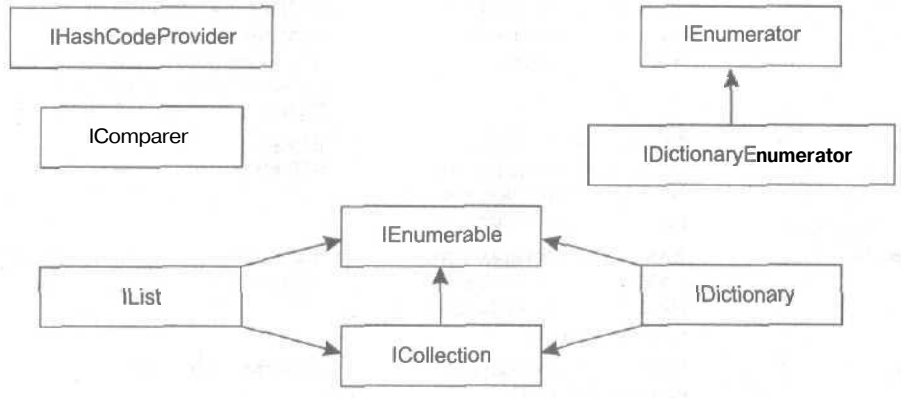


Рис. 4.10. Иерархия интерфейсов System.Collections

Таблица 4.2. Интерфейсы пространства имен System.Collections

Интерфейс	Назначение
ICollection	Определяет общие характеристики (например, только для чтения и т. д.) для класса-набора элементов
IComparer	Позволяет сравнивать два объекта
I Dictionary	Позволяет представлять содержимое объекта в виде пар имя — значение

продолжение

Таблица 4.2 (*продолжение*)

Интерфейс	Назначение
IDictionaryEnumerator	Используется для нумерации содержимого объекта, поддерживающего IDictionary
IEnumerable	Возвращает интерфейс IEnumerator для указанного объекта
IEnumerator	Обычно используется для поддержки конструкции foreach в отношении объектов
IHashCodeProvider	Возвращает хэш-код для реализации типа с применением выбранного пользователем алгоритма хэширования
IList	Обеспечивает методы для добавления, удаления и индексирования элементов в списке объектов

Наиболее часто используемые классы, определенные в пространстве имен System.Collections, представлены в табл. 4.3.

Таблица 4.3. Классы System.Collections

Класс	Назначение	Важнейшие из реализованных интерфейсов
ArrayList	Динамически изменяющий свой размер массив объектов	IList, ICollection, IEnumerable и ICloneable
Hashtable	Представляет набор взаимосвязанных ключей и значений, основанных на хэш-коде ключа	IDictionary, ICollection, IEnumerable и ICloneable. Кроме того, у типов, которые предназначены для хранения в Hashtable, всегда должен быть замещен метод System.Object.GetHashCode()
Queue	Стандартная очередь, реализованная по принципу FIFO (first-in-first-out, «первым пришел, перзым ушел»)	ICollection, ICloneable и IEnumerable
SortedList	Аналогично словарю, однако к элементам можно также обратиться по их порядковому номеру (индексу)	IDictionary, ICollection, IEnumerable и ICloneable
Stack	Очередь, реализованная по принципу LIFO (last-in-first-out, «последним пришел, первым ушел»), обеспечивающая возможности по проталкиванию данных в стек, выталкиванию данных из стека и считыванию данных	ICollection и IEnumerable

Пространство имен System.Collections.Specialized

Если ни один из классов, представленных в пространстве имен System.Collections, вам не подходит, есть смысл заглянуть в пространство имен System.Collections.Specialized. В этом пространстве имен определен свой набор типов для работы с наборами эле-

ментов. Как видно из названия пространства имен, эти типы предназначены для специальных случаев. В качестве примера можно назвать типы StringDictionary и ListDictionary, которые специальным образом реализуют интерфейс IDictionary. Мы настоятельно рекомендуем, прежде чем заняться собственной реализацией классов для работы с коллекциями элементов, заглянуть в электронную документацию Visual Studio.NET и познакомиться с уже готовыми классами. Вполне возможно, что значительная часть работы была уже сделана до нас.

Применение ArrayList

При ближайшем рассмотрении классов, определенных в пространстве имен System.Collections, выясняется, что они обладают очень схожей функциональностью и реализуют одни и те же интерфейсы. Поэтому вместо того, чтобы углубляться в подробности реализации каждого из классов, мы подробно разберем применение лишь одного из них — System.Collections.ArrayList. Надеемся, что после этого разобраться в использовании остальных классов не составит для вас особого труда.

В этой главе мы определили класс Cars, который представляет собой набор объектов типа Car. В нашем случае Cars был реализован на основе простого массива — то есть в качестве базового был использован класс System.Array. Поскольку у обычного массива возможностей не так много, нам пришлось создавать значительное количество дополнительного кода для того, чтобы можно было обращаться к элементам Cars из внешнего мира. Кроме того, при работе с Cars в его нынешнем состоянии мы столкнемся с серьезным ограничением — у этого массива фиксированный размер.

Гораздо удобнее и эффективнее было бы не делать лишнюю работу, а воспользоваться уже готовым к употреблению классом System.Collections.ArrayList, в котором многие нужные нам возможности реализованы изначально. Этот класс предоставляет в наше распоряжение готовые методы для вставки, удаления и нумерации внутренних элементов.

Для того чтобы воспользоваться возможностями ArrayList, мы применим не классическое наследование, а модель включения–делегирования, когда класс ArrayList будет вложен внутрь нашего класса Cars (почему именно так, выяснится чуть позже). Фактически единственное, что мы должны сделать, — реализовать в Cars набор открытых методов, которые будут передавать вызовы на выполнение различных действий (делегировать) внутреннему классу carList, производному от ArrayList. Выглядеть это может так:

```
// Нам больше не нужно реализовывать IEnumerator - все уже сделано за нас в ArrayList
public class Cars : IEnumerable
{
        // Это - тот самый внутренний класс, который и будет делать всю работу
        private ArrayList carList;

        // Создаем объект класса carList при помощи конструктора Cars
        public Cars() {carList = new ArrayList();}

        // Реализуем нужные нам методы для приема вызовов извне и передачи их carList

        // Метод для вставки объекта Car
```

```
        public void AddCar(Car c)
        { carList.Add(c); }

        // Метод для удаления объекта Car
        public void RemoveCar(int carToRemove)
        { carList.RemoveAt(carToRemove); }

        // Свойство, возвращающее количество объектов Car
        public int CarCount
        { get { return carList.Count; } }

        // Метод для очистки объекта - удаления всех объектов Car
        public void ClearAllCars()
        { carList.Clear(); }

        // Метод, который отвечает на вопрос - есть ли уже в наборе такой объект Car
        public bool CarIsPresent(Car c)
        { return carList.Contains(c); }

        // А все, что связано с реализацией IEnumerator, мы просто перенаправляем
        в carList
        public IEnumerator GetEnumerator()
        { return carList.GetEnumerator(); }
}
```

Помимо того что такое определение класса Cars само по себе проще, его еще и удобнее использовать:

```
// Применение нового варианта класса Cars
public static void MainC)
{
        Cars carLot = new Cars();

        // Чтобы было с чен работать, добавляем несколько объектов Car
        carLot.AddCar( new Car("Jasper", 200. 80));
        carLot.AddCar( new Car("Mandy", 140. 80));
        carLot.AddCar( new Car("Porker", 90, 90));
        carLot.AddCar( new Car("Jimbo", 40, 4));

        // Выводим информацию о каждом классе при помощи конструкции foreach
        Console.WriteLine("You have {0} in the lot: \n", carLot.CarCount);
        foreach (Car c in carLot)
        {
                Console.WriteLine("Name: {0}', c.PetName);
                Console.WriteLine("Max speed; {0}\n", c.MaxSpeed);
        }

        // Удаляем одну из машин
        carLot.RemoveCar(3);
        Console.WriteLine("You have {0} in the lot.\n", carLot.CarCount);

        // Добавляем еще одну машину и проверяем ее наличие в наборе
        Car temp = new Car("Zippy", 90. 90);
        carLot.AddCar(temp);

        if(carLot.CarIsPresent(temp))
                Console.WriteLine(temp.PetName + " is already in the lot.");

        // Убить их всех!
```

```
        carLot.ClearAllCars();
        Console.WriteLine("You have {0} in the lot.\n", carLot.CarCount");
}
```

Результат работы этой программы представлен на рис. 4.11.

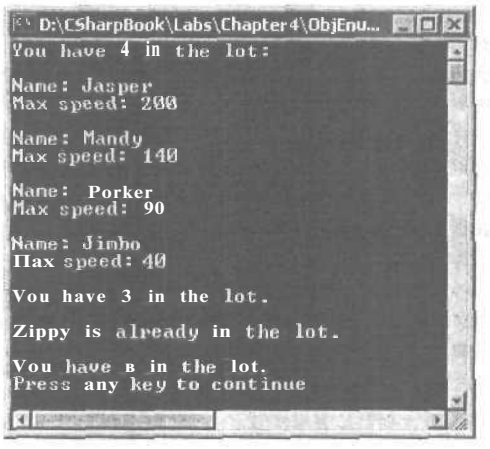


Рис. 4.11. Применение класса Cars, созданного на основе ArrayList

Теперь настало время ответить на вопрос — а почему нельзя просто произвести класс Cars от ArrayList? Зачем потребовалось все усложнять и создавать внутри Cars вспомогательный класс и дополнительные методы? Ответ очень прост — ArrayList сам по себе работает с любыми объектами. Это значит, что при использовании классического наследования класс Cars можно было бы заполнить объектами абсолютно любых типов C#. В этом легко убедиться:

```
ArrayList ar = new ArrayList();
ar.Add(carLot);
ar.Add("Hello");
ar.Add(new JamesBondCar());
ar.Add(23);
```

Чтобы как-то от этого защититься и запретить классу Cars содержать какие-либо объекты кроме объектов типа Car, мы и использовали модель включения-делегирования. Средством контроля того, что поступает в наш контейнерный класс Cars, стали методы этого класса.

Код приложения ObjectEnumWithCollectionможно найти в подкаталоге Chapter 4.

Подведение итогов

Если вы работали с COM, то эту главу вам было просто приятно читать. Интерфейс — это набор абстрактных методов, которые должны быть реализованы в классе, поддерживающем этот интерфейс. Поскольку интерфейс сам по себе не содержит никаких реализаций методов, лучше всего представлять себе интерфейс как некоторое поведение, которое должно поддерживаться реализующими этот интерфейс классами. Если два или более класса реализуют один и тот же интерфейс, то

вы сможете обращаться к объектам этих классов одинаково (таким образом достигается полиморфизм). В C# интерфейс определяется при помощи ключевого слова `interface`. Класс C# может поддерживать любое количество интерфейсов (при этом они просто перечисляются в объявлении класса через запятую). Интерфейсы могут вступать друг с другом в отношения наследования. При этом один интерфейс можно производить одновременно от нескольких базовых интерфейсов (в отличие от классов, для которых множественное наследование запрещено).

В библиотеках базовых классов .NET определено большое количество готовых стандартных интерфейсов. В этой главе мы рассмотрели наиболее важные интерфейсы, определенные в пространстве имен `System.Collections`. Вы можете использовать эти интерфейсы в своих пользовательских классах для того, чтобы реализовать в них поддержку необходимых вам возможностей, таких как клонирование объектов, сортировка и нумерация.

В завершение мы рассмотрели некоторые готовые классы для работы с наборами элементов, определенные в пространстве имен `System.Collections`, и выяснили, как можно достичь максимальной гибкости при использовании этих классов, применив метод включения–делегирования.

Дополнительные возможности классов C# 5

Эта глава посвящена нескольким исключительно полезным возможностям классов C#, без которых наш рассказ о классах был бы неполон.

Мы начнем со знакомства с методом индексатора (indexer method). Это средство C# позволяет вам создавать пользовательские классы-контейнеры, к внутренним объектам которых можно обращаться при помощи знакомых квадратных скобок ([]). Если у вас есть опыт работы с C++, то вы обнаружите, что создание метода индексатора в C# очень похоже на перегрузку оператора квадратных скобок в C++. После того как мы познакомимся с индексатором, логично будет рассмотреть вопросы, связанные с перегрузкой операторов в C# в целом.

Далее в этой главе будут рассмотрены три технологии, которые можно использовать для организации двустороннего взаимодействия объектов в нашем приложении. Первая технология — это применение делегатов, которые фактически: являются безопасными с точки зрения типов указателями на функции. Вторая технология — это протокол событий C#, который основан на использовании тех же делегатов. И наконец, мы рассмотрим вопросы, связанные с применением пользовательских интерфейсов для организации двустороннего взаимодействия (предмет, хорошо знакомый тем, кто пришел из мира COM).

В завершение главы мы рассмотрим документирование типов с использованием атрибутов XML, а также познакомимся с тем, как среда Visual Studio.NET может автоматически создавать web-документацию для ваших проектов. Несмотря на то что это вряд ли можно отнести к дополнительным возможностям классов C#, в полезности этих средств можно не сомневаться.

Создание пользовательского индексатора

Многие программисты давным-давно знакомы с тем, как можно обращаться к элементам контейнера при помощи *оператора индекса* (во многих языках программирования этот оператор выглядит как квадратные скобки — []).

```
// Объявляем массив целых чисел
int[] myInts = {10, 9, 100, 432, 9874};

// Применяем оператор индекса (квадратные скобки) для обращения к каждому
// из элементов массива
for(int j = 0; j < myInts.Length; j++)
    Console.WriteLine("Index [{0}] = {1}", j, myInts[j]);
```

В C# предусмотрены средства для создания пользовательских классов-контейнеров, к внутренним элементам которых можно обращаться при помощи того же оператора индекса, что и к элементам обычного массива встроенных типов. Метод, который обеспечивает такую возможность, получил название индексатора (indexer).

Прежде чем заниматься созданием этого метода, давайте вначале ознакомимся с тем, что он нам дает. Предположим, что мы уже добавили индексатор в класс Cars, с которым мы активно работали в предыдущей главе. В результате к элементам класса Cars (ими, как мы помним, могут быть только объекты Car) можно обращаться следующим образом:

```
// Индексатор позволяет обращаться к элементам контейнерного класса при помощи
// того же синтаксиса, что и к элементам обычного массива
public class CarApp
{
    public static void Main()
    {
        // Считаем, что в классе Cars уже реализован метод индексатора
        Cars carLot = new Cars();

        // Создаем несколько объектов Car и сразу добавляем их в carLot
        carLot[0] = new Car("FeeFee", 200, 0);
        carLot[1] = new Car("Clunker", 90, 0);
        carLot[2] = new Car("Zippy", 30, 0);

        // Выводим информацию о каждом внутреннем элементе в контейнере
        // на консоль:
        for(int i = 0; i < 3; i++)
        {
            Console.WriteLine("Car number [{0}]:", i);
            Console.WriteLine("Name: {0}", carLot[i].PetName);
            Console.WriteLine("Max speed: {0}", carLot[i].MaxSpeed);
        }
    }
}
```

То, что должно получиться при запуске этой программы, представлено на рис. 5.1.

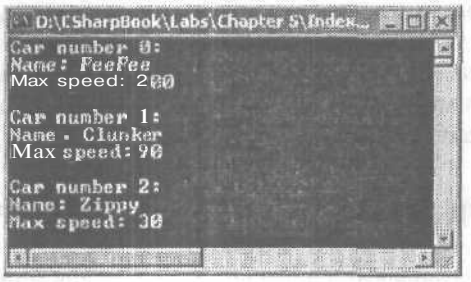


Рис. 5.1. Использование возможностей индексатора для обращения к элементам контейнерного класса

На этом примере мы могли убедиться, что классы с индексаторами по своим возможностям сильно напоминают классы, реализующие интерфейсы IEnumerator и IEnumerable. Главное различие между ними заключается в том, что для обращения к внутренним элементам этих классов-коллекций вместо использования ссылок на интерфейсы мы можем применить обычный оператор индекса — квадратные скобки, как при работе с обычным массивом.

А теперь главный вопрос: а как же все-таки снабдить класс Cars (или любой другой класс) возможностями индексатора? Ответ на этот вопрос — в коде, представленном ниже. Индексатор представляет собой слегка измененное свойство C#. В простейшем виде индексатор создается при помощи синтаксиса this[]:

```
// Добавляем в существующее определение класса индексатор
public class Cars: IEnumerator, IEnumerable
{
...
        // Вернемся к основан и будем использовать в качестве контейнера для объектов Car
        // обычный массив. Естественно, мы вправе, если так больше нравится, работать
        // с ArrayList
        private Car[] carArray;

        public Cars()
        {
                carArray = new Car[10];
        }

        // Индексатор позволяет обратиться к объекту Car по его порядковому номеру
        // в наборе (числовому индексу)
        public Car this[int pos]
        {
                // Метод для доступа к элементу в массиве
                get
                {
                        if(pos < 0 || pos > 10)
                                throw new IndexOutOfRangeException("Out of range!");
                        else
                                return (carArray[pos]);
                }
                // Метод для добавления новых объектов в массив
                set { carArray[pos] = value; }
                }
}
```

За исключением применения ключевого слова this индексатор ничем не отличается от обычного объявления свойства C#. Обратите внимание, что индексатор не обеспечивает никаких характерных для массивов C# возможностей, не считая применения оператора индекса. Например, мы не сможем воспользоваться имеющимся у каждого массива свойством Length:

```
// Используем System.Array.Length? Увы...
Console.WriteLine("Cars in stock: {0}", carLot.Length);
```

Если нам все-таки ужасно хочется использовать именно это свойство, придется создать его самостоятельно:

```
public class Cars
{
        ...
```

```
        public int Length() {// Код для получения информации о количестве элементов
                             // в массиве  }
}
```

Конечно же, гораздо проще не создавать заново подобные свойства, а просто воспользоваться возможностями одного из типов пространства имен System.Collections, в котором уже все реализовано.

Код приложения Indexer можно найти в подкаталоге Chapter 5.

Перегрузка операторов

В C#, как и в любом другом языке программирования, предусмотрен набор специальных символов для выполнения самых распространенных операций с внутренними типами данных. Например, многие программисты знают, что символ «+» можно использовать для получения одного числа из двух других:

```
// Оператор сложения (+) в действии
int a = 100;
int b - 240;
int c - a + b; // c = - 340
```

Надеюсь, что для вас этот код не стал откровением. Однако задумывались ли вы, почему тот же самый оператор «+» можно применять к любым встроенным типам C#, и не только числовым?

```
// Складываем два строковых значения
string s1 = "Hello";
string s2 - " world!";
string s3 - s1 + s2;  // s3 - - Hello world!
```

Ответ прост - оператор «+» перегружен таким образом, чтобы он мог нормально работать с самыми разными типами данных. Если мы применяем его к числовым типам, производится сложение. Если он применяется к строковым типам, производится слияние текстовых строк — конкатенация. Язык C# (подобно C++ и в противоположность Java) позволяет создавать пользовательские классы и структуры, которые будут реагировать по-своему (как мы определим) на те же самые встроенные операторы, например на оператор сложения «+». Этот прием называется перегрузкой операторов.

Рассматривать перегрузку операторов мы будем на примере класса Point (точка);

```
public class Point
{
        private int x, y;
        public Point(){}
        public Point(int xPos. int yPos)
        {
                x = xPos;
                y - yPos;
        }
        public override string ToString()
        {
                return "X pas: " +this.x + "Y pos: " + this.y;
        }
}
```

У нашей точки есть координаты, и мы вполне можем «складывать» между собой две точки (при этом будет происходить сложение их координат) и «вычитать» одну точку из другой (координаты второй точки будут вычитаться из первой). Удобнее всего, конечно, делать это так:

```
// «Складываем» и «вычитаем» точки
public static int Main(string[] args)
{
        // Задаем две точки
        Point ptOne - new Point(100, 100);
        Point ptTwo - new Point(40, 40);

        // «Складываем» две точки, чтобы получить третью
        Point bigPoint - ptOne + ptTwo;
        Console.WriteLine("Here is the big point: {0}", bigPoint.ToString());

        // «Вычитаем» одну точку из другой, чтобы получить третью
        Point minorPoint = bigPoint - ptOne;
        Console.WriteLine("Just a minor point: {0}", minorPoint.ToString());

        return 0;
}
```

Чтобы приведенный выше код заработал, осталось сделать самую малость — внести некоторые изменения в класс Point, чтобы объекты этого класса знали, *как* им реагировать на операторы «+» и «-». В C# для этого можно использовать ключевое слово operator. Обратите внимание, что его можно использовать только вместе с ключевым словом static:

```
// Класс Point с перегруженными операторами
public class Point
{
        private int x, y;
        public Point(){};
        public Point(int xPos, int yPos){ x - xPos; y - yPos; }

        // Перегружаем оператор сложения
        public static Point operator + (Point p1, Point p2)
        {
                Point newPoint - new Point(p1.x + p2.x, p1.y + p2.y);
                return newPoint;
        }

        // ...и вычитания
        public static Point operator - (Point p1, Point p2)
        {
                // Вычисляем значение новой координаты x
                int newX = p1.x - p2.x;
                if(newX < 0)
                        throw new ArgumentOutOfRangeException();

                // Вычисляем значение новой координаты y
                int newY - p1.y - p2.y;
                if(newY < 0)
                        throw new ArgumentOutOfRangeException();

                return new Point(newX, newY);
        }
```

```
    public override string ToStringO
    {
        return "X pos: " + this.x + " Y pos: " + this.y;
    }
}
```

Обратите внимание, что теперь наш класс содержит два немного странных с виду метода, которые называются operator + и operator -. Роль обоих методов понятна — вернуть новую точку (объект класса Point) с координатами, рассчитанными на основе координат двух исходных точек. В результате при сложении двух объектов Poi nt в коде программы будет неявно вызван метод operator +:

```
// p3 « Point.operator + (p1, p2)
p3 - p1 + p2;
```

То же самое относится и к вычитанию:

```
// p3 - Point.operator - (p1, p2)
p3 = p1 - p2:
```

Если вы запустите на выполнение программу с нашим примером, должен получиться результат, представленный на рис. 5.2.

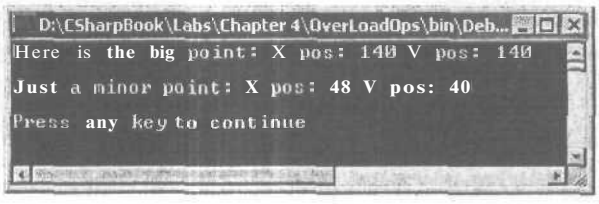


Рис. 5.2. Применяем перегрузку операторов

Перегрузка операторов может быть очень удобной, когда мы хотим заставить свои пользовательские типы реагировать на те же операторы, что и встроенные типы. Однако следует помнить, что подавляющее большинство языков программирования (и в их числе Java) перегрузку операторов не поддерживают. Требования обеспечить возможность перегрузки операторов нет и в Common Language Specification. Поэтому если вы работаете с большим проектом, разные модули которого созданы на разных языках, будьте готовы столкнуться с тем, что в некоторых модулях использовать перегрузку операторов будет невозможно.

Из этой ситуации есть очевидный выход — сделать так, чтобы каждому перегруженному оператору соответствовал обычный метод с «нормальным» названием. В модулях, написанных на языках, поддерживающих перегрузку операторов, будут использоваться соответствующие операторы, а в остальных модулях — самые обычные методы со стандартными названиями. Для нашего класса Point такое решение может выглядеть следующим образом:

```
// Кроне перегруженного оператора, в этом определении класса предусмотрен
// обычный метод с теми же возможностями
public class Point
{
...
    // Метод AddPoints работает точно так же, как перегруженный оператор сложения
    public static Point AddPoints (Point p1, Point p2)
```

```
        {
            return new Point(p1.x + p2.x, p1.y + p2.y);
        }

        // ...а метод SubtractPoints() - как перегруженный оператор вычитания
        public static Point SSubtractPoints (point p1, Point p2)
        {
            // Вычисляем значение новой координаты x
            int newX = p1.x - p2.x;
            if(newX < 0)
                throw new ArgumentOutOfRangeException();

            // Вычисляем значение новой координаты y
            int newY = p1.y - p2.y;
            if(newY < 0)
                throw new ArgumentOutOfRangeExceptionO:

            return new Point(newX, newY);
        }
}
```

Теперь точки можно складывать еще и так:

```
Point finalPt = Point.AddPoints(ptOne, ptTwo);
Console.WriteLine("My final point: {0}", finalPt.ToString());
```

Пожалуй, использование перегруженного оператора сложения более удобно и интуитивно понятно. Однако перегруженные операторы — это всего лишь более дружественная по отношению к пользователю класса разновидность открытого метода. Кроме того, не следует забывать о несовместимости этого средства с Common Language Specification. Поэтому при создании пользовательских классов, предназначенных для работы в реальных рабочих приложениях, всегда снабжайте перегруженные операторы обычными методами-аналогами. Это можно делать так, как было показано выше, однако, чтобы не дублировать код, еще удобнее просто передавать вызовы от перегруженного оператора обычному методу (или наоборот, если вам больше нравится). Выглядеть это может так:

```
public class Point
,{
...
        // Перегруженный оператор сложения вызывает обычный нетод
        public static Point operator + (Point p1, Point p2)
        {
            return AddPoints(p1, p2):
        }

        public static Point AddPoints (Point p1, Point p2)
        {
            return new Point(p1.x + p2.x, p1.y + p2.y);
        }
}
```

Перегрузка операторов равенства

Как мы помним, в C# часто возникает необходимость замещать метод System.Object.EqualsC), чтобы можно было выполнять сравнения структурных типов (в исходной реализации этот метод поддерживает только сравнение ссылочных типов),

Помимо замещения методов Equals() и GetHashCode() (при замещении Equals() замещение GetHashCode() обязательно) вам, скорее всего, потребуется заместить также и операторы равенства (= = и !=). Как это можно сделать, мы вновь продемонстрируем на примере с точкой.

```
// Новое воплощение класса Point снабжено еще и перегруженными операторами
// равенства - - и !=
public class Point
{
        public int x, y;
        public Point(){}
        public Point(int xPos, int yPos){x = xPos: y = yPos:}
...
        public override bool Equals(object o)
        {
                ifC ((Point)o).x = - this.x &&
                        ((Point)o).y - = this.y)
                                return true;
                else
                                return false;
        }

        public override int GetHashCodeC)
        { return this.ToString().GetHashCode(): }

        // А вот и сана перегрузка операторов равенства
        public static bool operator - =(Point p1, Point p2)
        {
                return p1.Equals(p2);
        }

        public static bool operator !=(Point pI, Point p2)
        {
                return !p1.Equals(p2);
        }
}
```

Обратите внимание, что для перегрузки операторов равенства нам ничего не надо выдумывать — мы просто передаем выполнение всей необходимой работы замещенному методу Equals(). Теперь, если нам потребуется использовать обновленный класс Point в нашей программе, мы можем сделать это следующим образом:

```
// Применяем перегруженные операторы равенства
public static int Main(string[] args)
{
...
        if(ptOne = = ptTwo)          // Две точки совпадают?
                Console.WriteLine("Same values!");
        else
                Console.WriteLine("Nope, different values.");

        if(ptOne != ptTwo)           // Это разные точки?
                Console.WriteLine("These are not equal.");
        else
                Console.WriteLineC'Same values!");
}
```

В подавляющем большинстве случаев программисты стремятся использовать именно понятные и знакомые операторы = = и !=. Применение их интуитивно, чего не скажешь о методе Equals О. Поэтому настоятельно рекомендуется в тех ситуациях, когда вы заместили метод Equals(}, позаботиться еще и о перегрузке операторов = = и !=.

Еще один момент, который необходимо обязательно отметить, — C# не позволит вам перегрузить оператор = = без перегрузки оператора ! = или наоборот (точно так же, как методы Equals() и GetHashCode() можно перегружать только в паре). Такой подход гарантирует, что все операторы равенства будут применяться одинаково корректно.

Код приложения OverLoadOps можно найти в подкаталоге Chapter 5.

Перегрузка операторов сравнения

В предыдущей главе мы познакомились с тем, как реализовывать интерфейс IComparable для того, чтобы было можно проводить сравнения объектов. Скорее всего, если мы реализовали этот интерфейс в определении класса, нам также понадобится перегрузить для этого класса операторы сравнения (<, >, <= и >=). Аргументы в пользу этого решения — такие же, как в случае перегрузки операторов равенства: использование привычных операторов проще и интуитивно понятнее, чем вызов для сравнения специальных методов. Необходимо отметить, что, как и в случае с операторами равенства, C# позволяет производить перегрузку операторов сравнения только парами. Первая пара — это операторы < и >, а вторая — <= и >=.

Если мы перегрузим операторы сравнения для класса Car, мы сможем работать с объектами этого класса следующим образом:

```
// Применяем перегруженный оператор < для объектов класса Car
public class CarApp
{
        public static int Main(string[] args)
        {
                // Создаем массив объектов класса Car
                Car[] my Autos = new Car[5];

                nyAutos[0] - new Car(123. "Rusty");
                nyAutos[1] * new Car(6. "Mary");
                nyAutos[2] = new Car(6. "Viper");
                nyAutos[3] = new Car(13, "NoName");
                nyAutos[4] = new Car(6. "Chucky");

                // Что меньше - Rusty или Chucky?
                if(myAutos[0] < myAutos[4])
                        Console.WriteLine("Rusty is less than Chucky!");
                else
                        Console.WriteLine("Chucky is less than Rusty!");
                return 0;
        }
}
```

Поскольку наш класс Car уже реализует интерфейс IComparable (мы сделали это в главе 4), то перегрузка операторов сравнения не представляет никакой сложности. Определение класса, в котором она проведена, может выглядеть следующим образом:

```
// Класс Car с перегруженными операторами сравнения
public class Car : IComparable
{
...
        public int CompareTo(object o)
        {
                Car temp = (Car)o;
                if(this.CarID > temp.CarID)
                        return 1;
                if (this.CarID < temp.CarID)
                        return -1;
                else
                        return 0:
        }
        public static bool operator < (Car c1, Car c2)
        {
                IComparable itfComp = (IComparable)c1;
                return (itfComp.CompareTo(c2) < 0):
        }
        public static bool operator > (Car cl. Car c2)
        {
                IComparable itfComp - (IComparable)c1;
                return (itfComp.CompareTo(c2) > 0);
        }
        public static bool operator <= (Car cl, Car c2)
        {
                IComparable itfComp = (IComparable)c1;
                return (itfComp.CompareTo(c2) <= 0);
        }
        public static bool operator >= (Car cl. Car c2)
        {
                IComparable itfComp = (IComparable)c1;
                return (itfComp.CompareTo(c2) >= 0);
        }
}
```

Код приложения ObjCompWithOps можно найти в подкаталоге Chapter 5.

Последние замечания о перегрузке операторов

Как мы смогли убедиться, C# обеспечивает возможность создавать типы, которые смогут реагировать нужным нам образом на хорошо знакомые стандартные операторы C#. Однако прежде, чем организовывать перегрузку операторов в каждом пользовательском классе C#, необходимо принять во внимание некоторые соображения.

Во-первых, перегружать оператор следует только тогда, когда применение этого оператора будет иметь вполне определенный смысл в случае некоторого класса. Например, предположим, что у нас есть класс Engine (двигатель). Если мы перегрузим для него оператор умножения, то что будет означать умножение одного двигателя на другой? Мне, например, это неясно. Как правило, обычно перегрузка операторов производится только для вспомогательных классов. Например, очевидными кандидатами на перегрузку операторов являются классы со строковыми значениями, классы для точек, прямоугольников, шестиугольников и т. п. Однако нет смысла перегружать какие-либо операторы для классов, моделирующих сотрудников, машины, наушники или бейсболки.

Во-вторых, нужно помнить, что не все языки платформы .NET поддерживают саму концепцию перегрузки операторов! Поэтому при работе над реальным приложением всегда следует придерживаться следующего правила — помимо перегруженного оператора в нашем классе обязательно должен быть «обычный» метод, выполняющий те же самые функции.

И последнее: не все операторы C# можно перегрузить, Информация о «перегружаемости» операторов представлена в табл. 5.1.

Таблица 5.1. Возможность перегрузки операторов C#

Оператор C#	**Можно** ли **произвести** перегрузку
+, -, !, ~, ++, --, true, false	Любой оператор из этого набора унарных операторов может быть перегружен
+, -, *, /, %, &, I/ ^, <<, >>	Любой оператор из этого набора бинарных операторов может быть перегружен
= =, !=, <, >, <=, >=	Все эти операторы сравнения могут быть перегружены. Однако следующие операторы могут быть перегружены только парами: > и <, <= и >=, = = и !=
[]	Этот оператор не может быть перегружен. Однако можно применять его к конкретному классу, если в нем реализован метод индексатора

Делегаты

К настоящему времени мы создали уже немало тестовых приложений. Практически все эти приложения состояли из блока `Main()`, в котором создавались различные объекты и им пересылались различные сообщения-команды. Однако мы пока не рассмотрели еще один очень важный аспект: как созданные объекты смогут посылать аналогичные сообщения тем объектам, которые их породили? В реальных приложениях, и в особенности при создании программ под Windows, необходимость в этом возникает очень часто.

При программировании под Windows на C и C++ основное средство для решения этой проблемы — это функция обратного вызова (callback function, или просто callback), которая основана на использовании указателей на функции в оперативной памяти. При помощи этого средства программист может обеспечить возможность обратного вызова (call back) одной функцией другой. Однако указатель на функцию — это всего лишь адрес в оперативной памяти, и из этого вытекает множество неудобств и потенциальных ошибок. Насколько было бы проще и безопаснее, если вместо «голого» адреса у нас была бы какая-то конструкция, которая могла бы проверять при выполнении обратного вызова и количество передаваемых параметров, и их тип, и возвращаемое значение, и следование определенной логике вызова... Однако в традиционных C и C++ ничего подобного нет.

А в C# такое средство есть! Оно называется *делегатом* (delegate) и выполняет те же действия, что и указатель на функцию, но гораздо более безопасными и лучше соответствующими принципам объектно-ориентированного программирования способами. При создании делегата в C# указывается не только имя метода, но и набор передаваемых функции параметров (если они есть), и возвращаемое функци-

ей значение. Как и все в мире C#, делегат — это специальный класс. Любой делегат производится от единого базового класса — System.MulticastDelegate с заранее определенным набором членов. Поэтому когда мы создаем новый делегат, например так:

```
public delegate void PlayAcidHouse(object PaulQakenfold. int volume):
```

в действительности в этот момент компилятор выполняет следующие команды по созданию нового класса:

```
public class PlayAcidHouse : System.MulticastDelegate
{
        PlayAcidHouse(object target, int ptr);

        // Синхронный метод Invoket)
        public void virtual Invoke(object PaulOakenfold. int volume);

        // Асинхронная версия того же самого обратного вызова
        public virtual IAsyncResult BeginInvoke(object PaulOakenfold. int volume,
                                                AsyncCallback cb, object o):
        public virtual void EndInvoke(IAsyncResult result):
}
```

Тот класс, который был создан при создании делегата, содержит два открытых метода Invoke() и BeginInvoke(), первый из которых предназначен для синхронного вызова, а второй — для асинхронного. Пока мы рассмотрим только средства для синхронного вызова в MulicastDelegate.

Пример делегата

Чтобы рассмотреть применение делегата на практике, нам потребуется внести в наш класс Car очередную порцию изменений. В этот раз мы добавим две новые переменные типа bool. Первая из них будет определять, является ли наш автомобиль грязным (переменная isDi rty), а вторая — нуждается ли автомобиль в замене покрышек (shouldRotate). Чтобы пользователю было проще взаимодействовать с новыми данными, в Car также будут добавлены дополнительные свойства и новый вариант конструктора:

```
// Класс Car вновь изменился
public class Car
(
...
        // Новые переменные!
        private bool isDirty; // Испачкан ли наш автомобиль?
        private bool shouldRotate; // Нужна ли замена шин?

        // Конструктор с новыми параметрами
        public car(string name. int max. int curr. bool dirty, bool rotate)
        {
                ...
                isOirty - dirty:
                shouldRotate = rotate:
        }
        // Свойство для isDirty
        public bool Dirty
        {
```

```
        get { return isDirty; }
        set { isDirty = value; }
    }
    // Свойство для shouldRotate
    public bool Rotate
    {
        get { return shouldRotate; }
        set { shouldRotate = value; }
    }
```

Теперь предположим, что мы объявили делегат в текущем пространстве имен следующим образом (вспомним, что делегат — это не более чем объектно-ориентированная надстройка, в основе которой — тот же указатель на функцию):

```
// Делегат - это класс, инкапсулирующий указатель на функцию. В нашем случае этой
// функцией должен стать какой-то метод, принимающий в качестве параметра объект класса
// Car и ничего не возвращающий;
public delegate void CarDelegate(Car c);
```

Если мы рассмотрим наше приложение при помощи ILDasm.exe, то мы обнаружим новый класс CarDelegate, который является производным от MulticastDelegate (рис. 5.3).

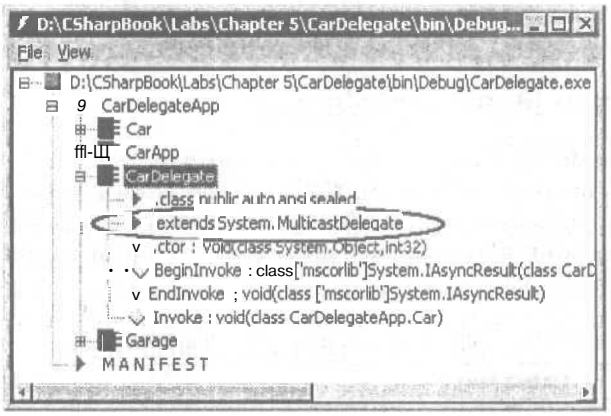


Рис. 5.3. Делегат в C# — это класс, производный от MulticastDelegate

Делегаты как вложенные типы

Сейчас созданный нами делегат существует отдельно от логически связанного с ним типа Car (оба они определены непосредственно в пространстве имен). Однако делегат можно поместить и непосредственно внутрь определения класса Car:

```
// Помещаем определение делегата анутрь определения класса
public class Car : Object
{
    // Теперь наш делегат получит служебное имя Car$carDelegate Сто есть станет
    // вложенным типом)
    public delegate void CarDelegate(Car c);
...
}
```

Поскольку, как мы выяснили, делегат — это новый класс, производный от System.MulticastDelegate, у нас получился вложенный класс CarDelegate. Убедиться в справедливости этого утверждения можно при помощи ILDasm.exe (рис. 5.4).

Рис. 5.4. Вложенный делегат

Члены System.MulticastDelegate

Конечно, делегат был создан как тип, производный от System.MulticastDelegate, не потому, что «так захотелось», а затем, чтобы каждый делегат C# унаследовал набор полезных членов, предназначенных для использования программистами. Наиболее интересные члены, которые наследуются любыми делегатами от System.MulticastDelegate, представлены в табл. 5.2.

Таблица 5.2. Некоторые унаследованные члены делегатов

Член	Назначение
Method	Это свойство возвращает имя метода, на который указывает делегат
Target	Если делегат указывает на метод — член класса, то этот член возвращает имя этого класса. Если Target возвращает значение типа null, то делегат указывает на статический метод
Combine()	Этот статический метод используется для создания делегата, указывающего на несколько разных функций
GetInvocationList()	Возвращает массив типов Delegate, каждый из которых представляет собой запись во внутреннем списке указателей на функции делегата
Remove()	Этот статический метод удаляет делегат из списка указателей на функции

Многоадресный делегат (multicast delegate) позволяет указывать на любое количество функций. При этом внутри делегата создается внутренний список указателей на функции. Поскольку все делегаты C# производятся от System.MulticastDelegate, то любой делегат C# потенциально является многоадресным. Чтобы добавить новый указатель на функцию во внутренний список делегата, используется метод

Combine() или перегруженный оператор сложения +, а чтобы удалить указатель ~ метод Remove().

Применение CarDelegate

Мы только что создали делегат, а значит, создали указатель на функцию. Это значит, что теперь мы сможем создавать новые функции, которые будут принимать наш делегат в качестве параметра. Лучше всего разобрать эту ситуацию на примере.

Предположим, что у нас есть новый класс, который называется Garage (гараж). Этот класс представляет собой набор объектов класса Car (для создания набора используется тип ArrayList). При создании объекта класса Garage внутрь этого объекта помещается несколько объектов класса Car.

Пусть наш класс Garage определяет открытый метод ProcessCars(), который принимает единственный параметр — делегат Car.CarDelegate. В определении метода ProcessCa rs() мы будем передавать все объекты Car из набора в качестве параметра той функции, на которую указывает наш делегат.

Чтобы проиллюстрировать схему внутренней работы делегатов, мы также воспользуемся возможностями двух унаследованных от System. Mul ti castDel egate членов — Target и Method. Они нам будут нужны для того, чтобы определить, на какую именно функцию в настоящий момент указывает делегат.

А вот и определение класса Garage:

```
// В классе Garage предусмотрен метод, принимающий CarDelegate в качестве параметра
public class Garage
{
    // Набор нашин в гараже
    ArrayList theCars - new ArrayList();

    // Создаем объекты машин в гараже
    public Garage()
    {
        // Применяем новый вариант конструктора
        theCars.Add(new car("Viper", 100. O, true, false));
        theCars.Add(new car("Fred", 100. 0. false. false));
        theCars.Add(new car("BillyBob", 100. 0. false. true));
        theCars.Add(new car("Bart", 100. 0. true. true));
        theCars.Add(new car("Stan", 100, 0. false. true));
    }

    // Этот метод принимает Car.CarDelegate в качестве параметра. Таким образом,
    // можно считать, что proc - это эквивалент указателя на функцию
    public void ProcessCars(Car.CarDelegate proc)
    t
        // Интересно, а куда мы передаем наш вызов?
        Console.WriteLine("***** Calling: {0} *****", proc.Method.ToString());

        // Еще одна проверка: вызываемый метод является статическим или
        // обычным?
        if(proc.Target !- null)
            Console.WriteLine("->Target: {0}", proc.Target.ToString());
        else
            Console.WriteLine("->Target is a static method");
```

```
            // Для чего это все затевалось: при помощи делегата вызываем метод
            // и передаем ему все объекты Car
            foreach(car c in the Cars)
                proc(c);
        }
}
```

При вызове метода ProcessCars() необходимо передать ему в качестве параметра имя метода, который должен обработать данный вызов. Например, пусть у нас есть два статических метода — WashCar() (помыть машину) и RotateTires() (заменить покрышки). Вызов этих методов через Car.CarDelegate может выглядеть следующим образом:

```
// Гараж передает право выполнить всю работу этим статическим функциям —
// наверное, у него нет хороших механиков...
public class CarApp
{
        // Первый метод, на который будет указывать делегат
        public static void WashCar(Car c)
        {
            if(c.Dirty)
                Console.WriteLine("Cleaning a car");
            else
                Console.WriteLine("This car is already clean...");
        }

        // Второй метод для делегата
        public static void RotateTires(Car c)
        {
            if(c.Rotate)
                Console.WriteLine("Tires have been rotated");
            else
                Console.WriteLine("Don't need to be rotated...");
        }

        public static int Main(string[] args)
        {
            // Создаем объект Garage
            Garage g - new Garage();

            // Моем все грязные машины
            g.ProcessCars(new Car.CarDelegate(WashCar));

            // Меняем шины
            g.ProcessCars(new Car.CarDelegate(RotateTires));

            return 0;
        }
}
```

Обратите внимание, что два наших статических метода в точности совпадают с сигнатурой, определенной делегатом (они принимают один объект типа Car и возвращают значение типа void — то есть ничего не возвращают). Когда мы передаем делегату имя функции, тем самым это имя добавляется во внутренний список указателей на функции для делегата. То, что должно получиться при запуске нашей программы, представлено на рис. 5.5. Обратите внимание на те сообщения, которые генерируются при помощи свойств Target и Method.

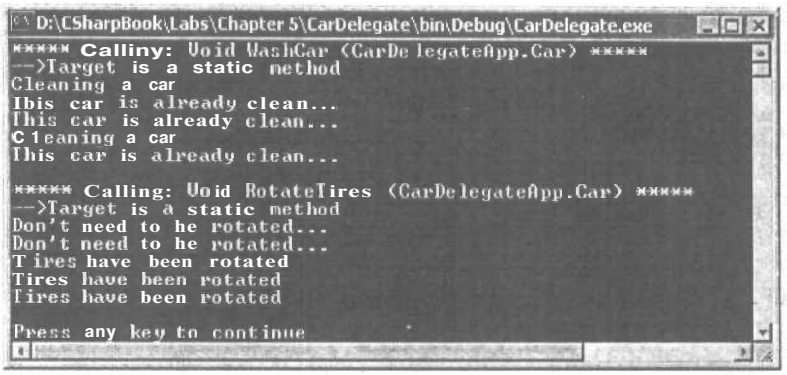


Рис. 5.5. Применение делегата

Анализ работы делегата

В нашем примере работа программы начинается с создания экземпляра объекта класса Garage. Далее при помощи делегата этот объект передает всю работу двум статическим функциям. Таким образом, если мы пишем:

```
// Вымыть все грязные машины
g.ProcessCars(new Car.CarDelegate(WashCar));
```

в действительности мы указываем: «Добавить указатель на функцию WashCar() во внутреннюю таблицу указателей делегата CarDelegate и передать этот делегат функции Garage.ProcessCar()». Как и большинстве настоящих центров техобслуживания, реальную работу выполняет другая часть системы (которая потом будет объяснять нам, почему замена масла, на которую нужно было потратить тридцать минут, заняла два часа). Таким образом, функция ProcessCar() реально работает следующим образом:

```
// CarDelegate уже указывает на функцию WashCar()
public void ProcessCars(Car.CarDelegate proc)
{
...
        foreach(Car c in the Cars)
                proc(c); // proc(c) => CarApp.WashCar(c)
...
}
```

То же самое можно сказать и в отношении второго метода:

```
// Поменять шины
g.ProcessCars(new Car.CarDelegate(RotateTires));
```

можно представить как

```
// Теперь CarDelegate указывает на функцию RotateTires
public void ProcessCars(Car.CarDelegate proc)
{
...
        foreach(Car c in the Cars)
                proc(c); // proc(c) => CarApp.RotateTires(c)
...
}
```

Обратите также внимание, что при вызове ProcessCars() мы обязаны создать и объект делегата при помощи ключевого слова new;

```
// Вымыть все грязные машины
g.ProcessCars(new Car.CarDelegate(WashCar));
// Поменять шины
g.ProcessCars(new Car.CarDelegate(RotateTires));
```

Если представлять себе делегат только как указатель на функцию, то это может показаться странным. Однако вспомним, что делегат — это на самом деле настоящий класс, производный от класса System.MulticastDelegate, и в этом свете создание нового объекта этого класса не должно вызывать удивления.

Многоадресность

Многоадресный делегат, как уже говорилось, — это объект, который может содержать в себе сразу несколько указателей на функции. В нашем примере мы не использовали такую возможность, а вместо этого два раза создавали делегат, каждый раз указывая лишь на одну функцию. Однако мы можем воспользоваться и многоадресностью:

```
// Добавляем во внутренний список указателей делегата сразу два указателя на функции:
public static int Main(string[] args)
{
        // Создаем объект Garage
        Garage g - new Garage();

        // Создаем два новых делегата
        Car.CarDelegate wash = new Car.CarDelegate(WashCar);
        Car.CarDelegate rotate - new Car.CarDelegate(RotateTires);
        // Чтобы объединить два указателя на функции в многоадресном делегате,
        // используется перегруженный оператор сложения (+). В результате создается новый
        // делегат, который содержит указатели на обе функции
        g.ProcessCars(wash + rotate);
        return G:
}
```

В этом примере вначале мы создали два отдельных делегата, каждый — с указателем на свою функцию. При вызове ProcessCars()этому методу в действительности передается новый делегат, который содержит все указатели на функции из двух предыдущих делегатов. Оператор + — это просто более удобный вариант статического метода Delegate.CombineO (см. табл. 5.2). Таким образом, тот же самый код мог выглядеть следующим образом:

```
// Оператор + можно использовать вместо метода Combine()
g.ProcessCars((Car.CarDelegate)Delegate.Combine(wash, rotate));
```

Если мы хотим, чтобы наш комбинированный делегат не исчез после выполнения метода ProcessCars(), а остался в боевой готовности для последующего использования, его можно создать за пределами ProcessCarsO:

```
// Создаем два новых делегата
Car.CarDelegate wash = new Car.CarDelegate(WashCar);
Car.CarDelegate rotate = new Car.CarDelegate(RotateTires);

// Объединяем их в новый делегат, который теперь можно использовать где угодно
MulticastDelegate d = wash + rotate;
```

```
// Передаем комбинированный делегат методу ProcessCars()
g.ProcessCars((Car.CarDelegate)d);
```

Вне зависимости от того, как именно мы создали многоадресный делегат, необходимо запомнить главное: добавление во внутреннюю таблицу нового указателя на функцию производится при помощи метода Combine() (или перегруженного оператора +), а удаление — при помощи статического метода Remove(). Первый параметр метода Remove() определяет делегат, с которым производится операция, а второй — тот указатель, который должен быть удален:

```
// Статический метод Remove() возвращает новый делегат - с удаленной записью
// в таблице указателей на функции
Delegate washOnly = MulticastDelegate.Remove(d, rotate);
g.ProcessCars((CarDelegate)washOnly);
```

Перед тем как запустить полученную программу на выполнение, мы вначале обновим ProcessCars() таким образом, чтобы при помощи метода Delegate.GetInvocationList() вывести на консоль все указатели на функции, хранящиеся во внутренней таблице. Этот метод возвращает массив объектов Delegate, которые мы выведем на консоль, используя конструкцию foreach:

```
// Выводим каждый член во внутренней таблице функций
public void ProcessCars (CarDelegate proc)
{
        // Куда мы передаем вызов?
        foreach(Delegate d in proc.GetInvocationList())
        {
                Console.WriteLine("***** Calling: " + d.method.ToString() + " *****");
        }
...
}
```

Результат выполнения нашей программы представлен на рис. 5.6.

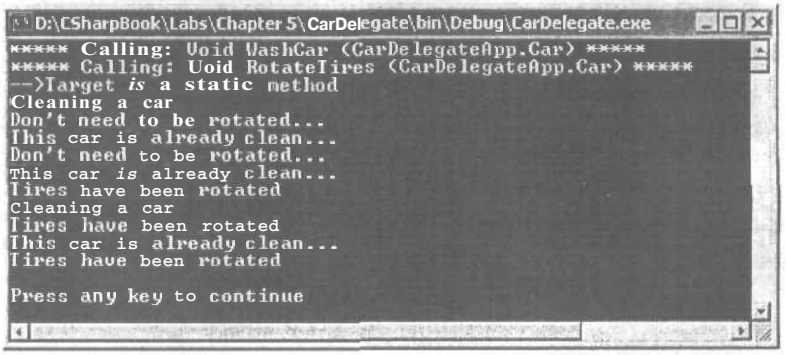


Рис. 5.6. Применение многоадресного делегата

Делегаты, указывающие на обычные функции

Во всех примерах, которые были приведены выше, делегаты указывали только на статические функции. Однако это совершенно не обязательно — мы вполне можем создавать делегаты, указывающие на самые обычные функции, вызов которых производится через объект класса. Для того чтобы показать это на примере,

мы переместим наши функции WashCar() и RotateTIres() в новый класс ServlceDept (отдел обслуживания);

```
// Статические функции перестали быть статическими и перенестились
// во вспомогательный класс
public class ServiceDept
{
        // Уже не статическая!
        public void WashCar(Car c)
        {
                if(c.Dirty)
                        Console.WriteLine("Cleaning a car");
                else
                        Console.WriteLine("This car is already clean...");
        }

        // То же самое
        public void RotateTires(Car c)
        {
                if(c.Rotate)
                        Console.WriteLine("Tires have been rotated");
                else
                        Console.WriteLine("Don't need to be rotated...");
        }
}
```

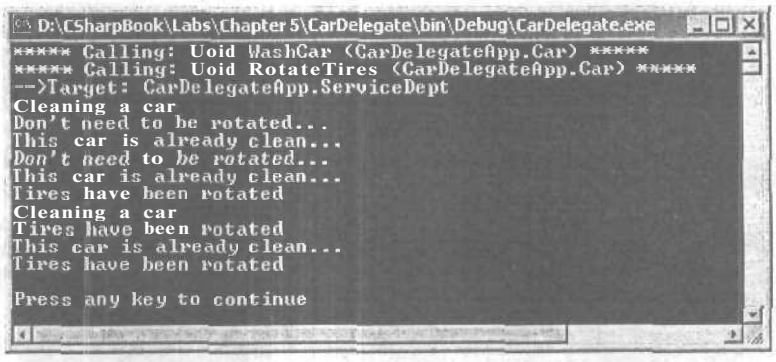


Рис. 5.7. Делегируем обычным методам

Теперь мы можем обновить наше приложение следующим образом:

```
// Делегаты будут указывать на обычные методы класса ServiceDept
public static int Main(string[] args)
{
        // Создаем гараж
        Garage g = new Garage();

        // Создаем отдел обслуживания
        ServlceDept sd = new ServiceDept();

        // Гараж делегирует работу отделу обслуживания
        Car.CarDelegate wash = new Car.CarDelegate(sd.WashCar);
        Car.CarDelegate rotate - new Car.CarDelegate(sd.RotateTires);
        MulticastDelegate d = wash + rotate;

        // Обращаемся в гараж с просьбой сделать эту работу
```

```
        g.ProcessCars((Car.CarDelegate)d);

        return 0;
}
```

То, что должно получиться, представлено на рис. 5.7 — обратите внимание на имя вызываемого метода.

Код приложения CarDelegate можно найти в подкаталоге Chapter 5.

События

Делегаты — это, безусловно, очень полезная синтаксическая конструкция C#, которая позволяет определять имя вызываемой функции во время выполнения, а не при компиляции. Однако еще более важная роль делегатов заключается в том, что на них основана модель событий C#.

Чаще всего события (events) используются в приложениях под Windows с графическим интерфейсом пользователя, в которых такие элементы управления, как Button (кнопка) или Calendar (календарь), реагируя на события, выдают информацию на той же панели, где они расположены. В качестве примера такого события можно привести, например, щелчок мышью на кнопке. Однако применение событий вовсе не ограничено приложениями с графическим интерфейсом — они могут быть исключительно полезными и в обычных консольных программах, как мы убедимся в наших примерах.

Как мы помним, класс Car (а точнее, его метод Car.SpeedUp()) в его текущем состоянии приводит к исключению при попытке увеличить скорость уже вышедшего из строя автомобиля. Однако исключение — это очень сильное средство (если оно не перехватывается должным образом, выполнение программы будет прервано), и если есть возможность обойтись без исключения, лучше попробовать поступить именно так. Правильнее в этой ситуации будет использовать вместо исключения пользовательское событие, возникающее при переходе автомобиля в нерабочее состояние.

Мы изменим наш класс Car, добавив в него два события. Первое событие (AboutToBlow) будет происходить тогда, когда текущая скорость всего на 10 миль в час меньше максимально допустимой. Второе событие (Exploded) будет возникать тогда, когда пользователь пытается ускорить автомобиль, который уже вышел из строя. Создание любого события в C# состоит из двух этапов. Первый этап — создание делегата, который будет использован для вызова нужного нам метода при срабатывании события, а второй этап — определение собственно события при помощи ключевого слова event. Новый вариант класса Car может выглядеть следующим образом:

```
// Этот класс Car будет посылать пользователю сообщения о своем состоянии
public class Car
{
...
        // Переменная для хранения информации о состоянии машины
        private bool dead;

        // Делегат. Он нужен, чтобы вызвать функцию или функции при возникновении
        // события
        public delegate void EngineHandler(string msg);
```

```
        // Два события
        public static event EngineHandler Exploded;
        public static event EngineHandler AboutToBlow;
...
}
```

Организовать срабатывание события (то есть организовать отправку этого события тому, кто ожидает его возникновения) очень просто — достаточно указать имя события и все необходимые параметры, которые требует соответствующий делегат. Например, новый вариант метода SpeedUp(), в котором логика работы с исключениями заменена на логику вызова события, может выглядеть так:

```
// Вызываем нужное событие а заивисимости от состояния объекта Car
public void SpeedUp(int delta)
{
        // Если автомобиль уже вышел из строя, генерируем событие Exploded
        if(dead)
        {
                if(Exploded != null)
                        Exploded("Sorry, this car is dead...");
        }
        else
        {
                currSpeed += delta;

        // Приближаемся к опасной черте? Генерируем событие AboutToBlow
        if(10 = - maxSpeed - currSpeed)
                if(AboutToBlow != null)
                        AboutToBlow("Careful, approaching terminal speed!");

        // Все нормально! Работаем как обычно
        if(crrSpeed >= maxSpeed)
                dead = true;
        else
                Console.WriteLine("\tCurrSpeed = {0}", currSpeed);
        }
}
```

Теперь мы настроили класс Car таким образом, чтобы при соответствующих условиях генерировались два события. Применение нового класса мы рассмотрим чуть позже — сразу же после того, как обсудим архитектуру модели событий чуть более подробно.

Как работают события

Любое событие — это на самом деле набор двух скрытых методов, определенных как public. Один из методов начинается с приставки add_, а второй — с приставки remove_. Например, событие Exploded отображается в следующие методы:

```
// Это событие отображается в скрытые методы *
// add_Exploded() и
// remove_Exploded()
//
public static event EngineHandler Exploded:
```

Помимо скрытых методов add_XXXX() и remove_XXXX(), каждому событию также соответствует статический класс, определенный как private. Его назначение —

привязывать событие к соответствующему делегату. При этом при срабатывании события будет вызван каждый из методов делегата. Такой способ позволяет сразу нескольким «приемникам событий» (event sinks) получать одно-единственное происшедшее событие.

Увидеть, что на самом деле кроется за нашими событиями, можно на рис. 5.8,

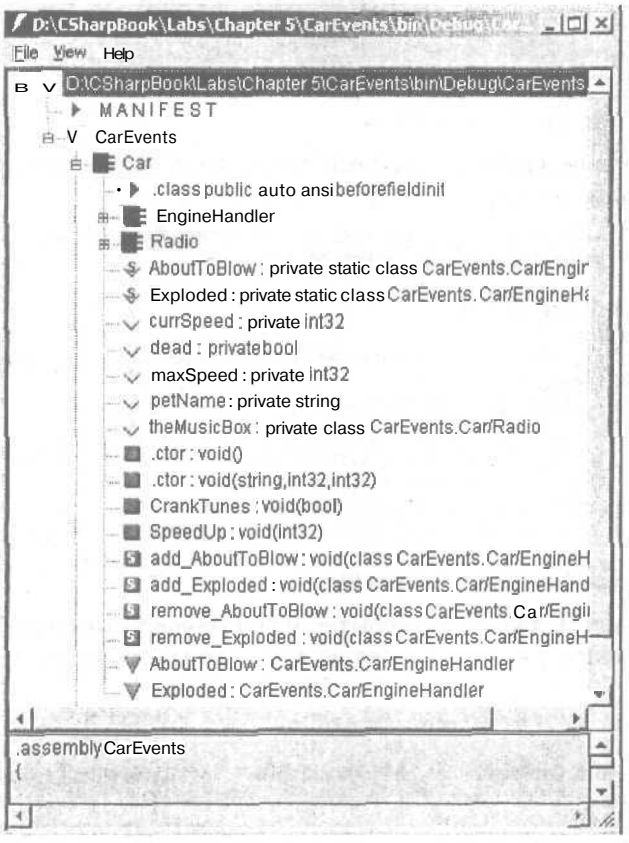


Рис. 5.8. События C# и то, что им соответствует

Таким образом, любое событие (в нашем случае событий два — Exploded и About ToBlow) внутренне представляется следующими членами:

- статическим классом, определенным как private;
- методом add_XXXX();
- методом remove_XXXX().

Если мы захотим посмотреть, какие инструкции промежуточного языка кроются за методом add_AboutToBlow, то обнаружим следующий код (обратите внимание на вызов метода Delegate.Combine()):

```
.method public hidebysig specialname static
void add_AboutToBlow(class CarEvents.Car/EngineHandler 'value') cil managed synchronized
```

```
{
        // Code size 22 (0x16)
        .maxstack 8
        IL_0000: ldsfld class CarEvents.Car/EngineHandler CarEvents.Car::AboutToBlow
        IL_0005: ldarg.0
        IL_0006: call class [mscorlib]System.Delegate
                            [mscorlib]System.Delegate::Combine(class
                            [mscorlib]System.Delegate, class
                            [mscorlib]System.Delegate)
        IL_000b: castclass CarEvents.Car/EngineHandler
        IL_0010: stsfld class CarEvents.Car/EngineHandler CarEvents.Car::AboutToBlow
        IL_0015: ret
} // Конец метода Car::add_AboutToBlow
```

Как вы, наверное, уже догадываетесь, `remove_AboutToBlow` производит аналогичный вызов метода `Delegate.Remove()`:

```
.method public hidebysig specialname static void remove_AboutToBlow(class CarEvents.Car/
EngineHandler 'value') cil managed synchronized
{
        // Размер кода 22 (0x16)
        .maxstack 8
        IL_0000: ldsfld class CarEvents.Car/EngineHandler CarEvents.Car::AboutToBlow
        IL_0005: ldarg.0
        IL_0006: call class [mscorlib]System.Delegate
                            [mscorlib]System.Delegate::Remove(class
                            [mscorlib]System.Delegate, class
                            [mscorlib]System.Delegate)
        IL_000b: castclass CarEvents.Car/EngineHandler
        IL_0010: stsfld class CarEvents.Car/EngineHandler CarEvents.Car::AboutToBlow
        IL_0015: ret
} // Конец метода Car::remove_AboutToBlow
```

В инструкциях IL для самого события можно обнаружить теги `[.addon]` и `[.removeon]` для методов `add_XXXX()` и `remove_XXXX()` со ссылкой на служебный класс `EngineHandler`:

```
.event CarEvents.Car/EngineHandler AboutToBlow
{
        .addon void CarEvents.Car::add_AboutToBlow(class CarEvents.Car/EngineHandler)
        .removeon void CarEvents.Car::remove_AboutToBlow(class CarEvents.Car/
                                                         EngineHandler)
} // Конец события Car::AboutToBlow
```

Теперь, когда мы познакомились с тем, как создавать события, самое время узнать, как эти события можно принимать.

Прием событий

Предположим, что мы создали объект класса `Car` и теперь наша задача — организовать реакцию на события, которые этот объект будет посылать. Если подумать, то задача заключается в создании метода, представляющего приемник события, — то есть метода, вызываемого делегатом. Формулируем задачу еще конкретнее — нам необходимо вызвать нужный вариант метода `add_XXXX()`, чтобы добавить наш принимающий метод в таблицу указателей на функции в делегате, с которым связано событие. Однако в C# вызвать скрытые методы `add_XXXX()` и `remove_XXXX()` напрямую запрещено — это можно делать только при

помощи перегруженных операторов += и -=. Поэтому «подключить» приемник для прослушивания событий можно только при помощи следующего синтаксиса:

```
// Начинаем прослушивание
// ObjectVariable.EventName += new ObjectVariable.DelegateName(functionToCall);
//
Car.Exploded += new Car.EngineHandler(OnBlowUp);
```

Если мы хотим «отключиться» от прослушивания событий, то, конечно, для этого нужен перегруженный оператор -=:

```
// Прекращаем прослушивание:
// ObjectVariable.EventName -- new ObjectVariable.DelegateName(functionToCall);
//
Car.Exploede -- new Car.EngineHandler(OnBlowUp);
```

Полный пример реализации реакции на события приведен ниже:

```
// Создаем объект Car и настраиваем реакцию на события, которые этот класс инициирует
public class CarApp
{
        public static int Main(string[] args)
        {
                Car c1 - new Car("SlugBug", 100, 10):

                // Устанавливаем приемники событий
                Car.Exploded += new Car.EngineHandler(OnBlowUp);
                Car.AboutToBlow +- new Car.EngineHandler(OnAboutToBlow);

                // Разгоняем машину (при этом будут инициированы события)
                for(int 1 - 0: 1 < 10: i++) c1.SpeedUp(20);

                // Отключаем приемники событий
                Car.Exploded -= new Car.EngineHandler(OnBlowUp);
                Car.AboutToBlow -- new Car.EngineHandler(OnAboutToBlow);

                // Теперь реакции на события нет!
                fordnt i = 0; i < 10: i++) c1.SpeedUp(20);
                return 0:
        }

        // Приемник OnBlowUp
        public static void OnBlowUp(string s)
        {
                Console.WriteLine("Message from car: {0}", s):
        }

        // Приемник OnAboutToBlow
        public static void OnAboutToBlow (string s)
        {
                Console.WriteLine("Message from car: {0}", s):
        }
}
```

Результат работы этой программы представлен на рис. 5.9.

Если мы хотим, чтобы какое-то событие вызывало срабатывание сразу нескольких приемников, надо просто добавить необходимое их количество:

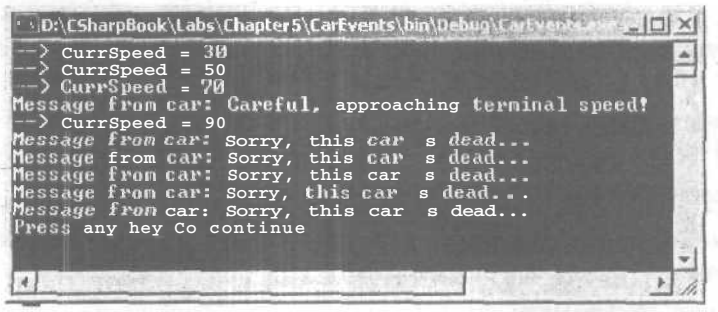


Рис. 5.9. Управляем набором событий для объекта Car

```
// Несколько приемников событий
public class CarApp
{
        public static int Main(string[] args}
        {
                // Создаем объект класса Car пак обычно
                Car c 1 = new Car("SlugBug", 100, 10):

                // Устанавливаем приемники событий:
                Car.Exploded += new Car.EngineHandler(OnBlowUp);
                Car.Exploded += new Car.EngineHandler(OnBlowUp2);
                Car.AboutToBlow += new Car.EngineHandler(OnAboutToBlow);

                // Разгоняем машину (при этом будут инициированы события)
                for(int i = 0: i < 10: i++)
                        c1.SpeedUp(20);

                // Отключаем приемники событий
                Car.Exploded -= new Car.EngineHandler(OnBlowUp);
                Car.Exploded -= new Car.EngineHandler(OnBlowUp2); .
                Car.AboutToBlow -= new Car.EngineHandler(OnAboutToBlow);

                ...
        }

        // Первый приемник события Exploded
        public static void OnBlowUp(string s)
        {
                Console.WriteLine("Message from car: {0}", s):
        }

        // Второй приемник события Exploded
        public static void OnBlowUp2(string s)
        {
                Console.WriteLine("-> AGAIN I say: {0}", s):
        }

        // Приемник для события AboutToBlow
        public static void OnAboutToBlow(string s)
        {
                Console.WriteLine("Message from car: {0}", s):
        }
}
```

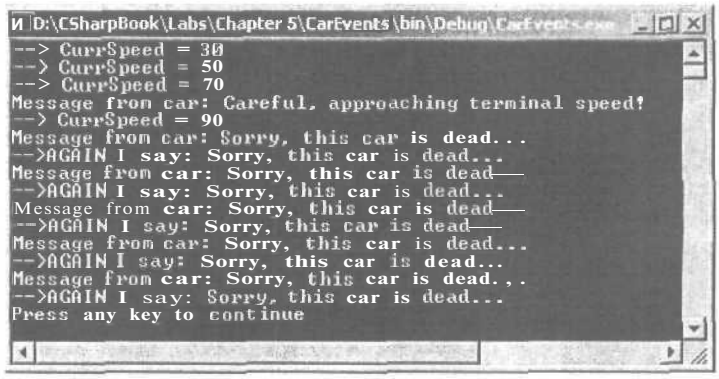


Рис. 5.10. Работа с несколькими приемниками единственного события

Теперь при возникновении события Exploded связанный с этим событием делегат вызовет и метод OnBlowUp(), и метод OnBlowUp(). Убедиться в этом можно, посмотрев на рис. 5.10.

Объекты как приемники событий

К этому моменту мы выяснили, как создавать объекты, участвующие в двусторонних взаимодействиях при помощи делегатов и событий, В приведенных выше примерах методы, работающие в качестве приемников событий, были помещены непосредственно в класс CarApp вместе с другим содержимым. Однако с точки зрения инкапсуляции и более четкой организации есть смысл создать вспомогательный класс, в котором будут находиться только методы-приемники событий. Такой класс: может выглядеть следующим образом:

```
// Служебный класс для приемников событий
public class CarEventSink
{
        // Приемник OnBlowUp для события Exploded
        public void OnBlowUp(string s)
        {
                Console.WriteLine("Message from car: {0}", s);
        }

        // Приемник OnBlowUp2 для того же события
        public void OnBlowUp2(string s)
        {
                Console.WriteLine(""->AGAIN I say: {0}", s);
        }

        // Приемник OnAboutToBlow для события AboutToBlow
        public void OnAboutToBlow(string s)
        {
                Console.WriteLine("Message from car: {0}", s);
        }
}
```

Такая организация, при которой методы-приемники событий удалены из CarApp, представляется более аккуратной. Работа с этими приемниками будет производиться через отдельный объект класса CarEventSink:

```
// Обрат/те внимание на создание объекта CarEventSink и его использование
public class CarApp
{
      public static int Main(string[] args)
      {
            Car c1 - new Car("SlugBug", 100, 10);

            // Создаем объект с приемниками
            CarEventSink sink = new CarEventSink();

            // Устанавливаем приемники
            Car.Exploded += new Car.EngineHandler(sink.OnBlowUp);
            Car.Exploded += new Car.EngineHandler(sink.OnBlowUp2);
            Car.AboutToBlow +- new Car.EngineHandler(sink.OnAboutToBlow);

            for(int i - 0; i < 10; i++)
                  c1.SpeedUp(20);

            // Отключаем приемники событий
            Car.Exploded -= new Car.EngineHandler(sink.OnBlowUp);
            Car.Exploded -= new Car.EngineHandler(sink.OnBlowUp2);
            Car.AboutToBlow -= new Car.EngineHandler(sink.OnAboutToBlow);

            return 0;
      }
}
```

Результат работы этой программы будет тем же, что и на рис. 5.10. Код приложения CarEvents можно найти в подкаталоге Chapter 5.

Реализация обработки событий с использованием интерфейсов

Программистам COM хорошо знакома концепция интерфейсов обратного вызова (callback interfaces). Это средство позволяет клиенту COM получать события от сокласса при помощи пользовательского интерфейса COM. Преимущество такого подхода заключается в том, что ресурсов расходуется меньше, чем при использовании официально рекомендованной в COM архитектуры точки стыковки. Те же самые интерфейсы обратного вызова можно использовать и в C# (и во всей .NET). Материал этого раздела в дополнение к уже освоенному открывает перед нами другие возможности и подтверждает представления о том, что любую проблему можно решить несколькими разными способами.

Реализовывать интерфейс обратного вызова мы будем на примере все того же класса Car. Пусть условия останутся прежними — этот класс должен как-то передать сообщения внешнему миру при наступлении двух событий: когда автомобиль должен вот-вот развалиться (его скорость всего на 10 миль в час меньше максимально допустимой) и когда это печальное событие уже произошло. Однако сейчас мы не будем использовать ни делегат, ни событие. Вместо этого мы создадим следующий пользовательский интерфейс:

```
// Интерфейс для работы с событиями
public interface IEngineEvents
{
      void AboutToBlow(string msg);
```

```
        void Exploded(string msg);
}
```

Если у нас есть интерфейс, то должен быть и класс, который его реализует. Далее мы создадим объект этого класса, и методам этого объекта и будет передавать вызовы автомобиль (объект класса Car) при наступлении событий. А класс, реализующий интерфейс IEngineEvents, может выглядеть следующим образом:

```
// Класс с методами-приемниками событий
public class CarEventSink : IEngineEvents
{
        public void AboutToBlow(string msg)
        {
                Console.WriteLine(msg);
        }

        public void Exploded(string msg)
        {
                Console.WriteLine(msg);
        }
}
```

Теперь, когда мы можем создать объект с методами, — приемниками событий, наша следующая задача — передать соответствующую ссылку (это будет ссылка на интерфейс) объекту класса Car. Объект Car будет хранить эту ссылку и сможет произвести по ней обратный вызов в случае наступления ожидаемых нами событий. Чтобы класс Car мог это сделать, нам потребуется внести в него некоторые изменения.

Первое, что нам потребуется в классе Car, — это метод, принимающий ссылку на интерфейс. В соответствии с традициями COM-программирования мы назовем этот метод Advise(). К этому методу необходимо добавить парный метод, на случай если нам потребуется отключиться от источника событий и перестать обращать на них внимание. Опять-таки в соответствии с традициями COM этот метод получит имя Unadvise(). Ну, и наконец, чтобы функциональность при использовании интерфейсов для обработки событий была не меньше, чем при использовании стандартных делегатов, мы должны позаботиться о том, чтобы можно было хранить сколько угодно приемников для событий. Лучше всего для этой роли подойдет объект класса ArrayList — массив переменной длины, обладающий массой полезных членов.

Новые члены класса Car будут выглядеть следующим образом:

```
// Этот вариант класса Car не использует ни делегатов C#, ни событий
// в их обычном понимании
public class Car
{
        // Хранилище для подключенных приемников
        ArrayList itfConnections = new ArrayList();

        // Метод для подключения приемников
        public void Advise(IEngineEvents itfClientImpl)
        {
                itfConnections.Add(itfClientImpl);
        }

        public void Unadvise(IEngineEvents itfClientImpl)
```

```
        {
            itfConnections.Remove(itfClientImpl);
        }
```

Теперь необходимо внести изменения и в метод Car.SpeedUp(), чтобы он при помощи цикла вызывал при наступлении соответствующего события все нужные методы в массиве itfConnections:

```
// Протокол обработки событий, основанный на интерфейсах:
//
class Car
{
...
    public void SpeedUp(int delta)
    {
        // Если машина развалилась, посылаем сообщение о событии Exploded каждому
        // приемнику для этого события
        if(dead)
        {
            foreach(IEngineEvents e in itfConnections)
                e.Exploded("Sorry, this car is dead...");
        }
        else
        {
            currSpeed += delta;

            // Если автомобиль только близок к худшему исходу, работаем
            // с другим событием
            if(10 * maxSpeed - currSpeed)
            {
                foreach(IEngineEvents e in itfConnections)
                    e.AboutToBlow("Careful buddy! Gonna blow!");
            }

            // А у нас все в порядке
            if(currSpeed>=maxSpeed)
                dead = true;
            else
                Console.WriteLine("\tCurrSpeed = {0}", currSpeed);
        }
}
```

Клиентский код, использующий интерфейс обратного вызова для реагирования на события класса Car, может выглядеть так:

```
// Создаем объект класса Car и реагируем на его события
public class CarApp
{
    public static int Main(string[] args)
    {
        Car c1 = new car("SlugBug", 100, 10);

        // Создаем объект с методами-приемниками
        CarEventSink sink = new CarEventSink();

        // Передаем объекту класса Car ссылку на объект с методами-приемниками
```

```
            c1.Advise(sink);

            // Разгоняем автонобиль, чтобы наступили события
            for(int i=0; i < 10; i++)
                  c1.SpeedUp(20);

            // Отключаем приемник
            c1.Unadvise(sink);
            return 0;
      }
}
```

Результат работы программы (рис. 5.11) выглядит очень знакомо.

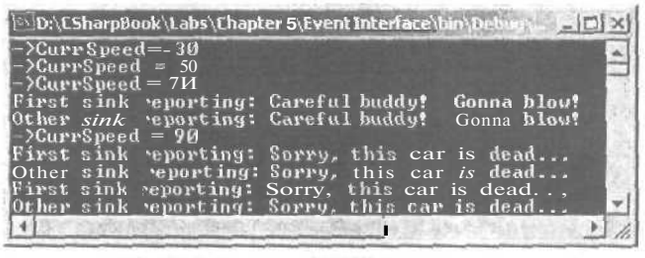


Рис. 5.11. Обработка событий при помощи интерфейсов

Код приложения EventInterface можно найти в подкаталоге Chapter 5.

Документирование в формате XML

Последняя часть этой главы посвящена автоматизированному составлению документации в формате XML средствами C#.

Если у вас есть опыт работы с Java, то вы, без сомнения, знакомы с утилитой javadoc, при помощи которой можно преобразовать код Java в представление в формате HTML. Модель автоматизированного документирования, принятая в C#, выглядит немного по-другому: во-первых, в результате получается файл в формате XML (хотя есть средства и для преобразования в HTML), а во-вторых, создание этого файла XML производится непосредственно компилятором C#, а не отдельной утилитой.

У вас может возникнуть вопрос: а почему XML, а не обычный HTML? Ответ прост — у формата XML гораздо больше возможностей. Мы можем программным образом применять преобразования к коду XML (это гораздо проще, чем работать таким же образом с более свободным HTML) или, к примеру, считывать при помощи модулей библиотек базовых классов информацию из файла XML непосредственно в базу данных.

Для составления документа в формате XML используются непосредственно файлы с исходным кодом проекта. Конечно же, при этом нам необходимо будет добавить в наш проект некоторые теги XML — и C# предоставляет такую возможность. Теги XML должны записываться как комментарии особого типа — через тройной слэш (///), в отличие от комментариев в стиле C++ (//) и C (// ...). После тройного слэша можно помещать любые теги XML, включая набор заранее готовых для документирования кода тегов, представленный в табл. 5.3.

Таблица 5.3. Некоторые теги XML

Теги XML, используемые для документирования кода	Назначение
<c>	Указывает, что текст внутри описания должен быть помечен как код
<code>	Указывает, что несколько строк должны быть помечены как код
<example>	Помечает пример кода для описываемого элемента
<exception>	Помечает исключения, которые может генерировать данный класс
<list>	Используется для вставки списка в файл документации
<param>	Описывает параметр
<paramref>	Связывает указанный тег XML с конкретным параметром
<permission>	Используется для документирования информации о разрешениях на доступ к какому-либо члену
<remarks>	Используется для описания конкретного члена
<returns>	Описывает значение, возвращаемое членом
<see>	Используется для перекрестных ссылок в описании
<seealso>	Для создания раздела «см. также» (see also) в описании
<summary>	Для создания раздела «итоги» в описании
<value>	Используется для документирования указанного свойства

Упрощенный вариант класса Car с некоторыми тегами XML представлен ниже. Советуем обратить внимание на применение тегов <summary> и <paramref>:

```
/// <summary>
/// Это - простой вариант класса Car, на примере которого
/// демонстрируется работа с документацией в формате XML
/// </summary>
public class SimpleCar
{
        /// <summary>
        /// Есть ли у вас в машине солярий?
        /// </summary>
        private bool hasSunroof - false:

        /// <summary>
        /// Этот специальный вариант конструктора используется,
        /// чтобы определить наличие солярия
        /// </summary>
        /// <param name="hasSunroof"> </param>
        public SimpleCar(bool hasSunroof)
        {
                this.hasSunroof - hasSunroof;
        }

        /// <summary>
        /// Этот метод позволяет открыть солярий
        /// в вашей машине
        /// </summary>
        /// <param name="state"> </param>
        public void OpenSunroof(bool state)
```

```
        {
            if(state = = true && hasSunroof = = true)
            {
                Console.WriteLine("Put sunscreen on that bald head!");
            }
            else
            {
                Console.WriteLine("Sorry...you don't have a sunroof.");
            }
        }
        /// <summary>
        /// Точка входа для приложения
        /// </summary>
        public static void Main()
        {
            SimpleCar c = new SimpleCar(true);
            c.OpenSunroof(true);
        }
    }
```

После того как мы разместим необходимые теги в исходном коде, можно приступать к изготовлению файла документации XML. Для этого нам потребуется запустить компилятор C#, указав имя создаваемого файла XML после параметра /doc:. Выглядеть это может так:

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

Конечно же, создавать документацию в формате XML можно и при помощи интегрированной среды разработки Visual Studio.NET. Для этого вначале просто нажмем кнопку Properties (Свойства) в окне Solution Explorer (рис. 5.12). Откроется окно Project Properties (Свойства проекта). Нам необходимо открыть папку Configuration Properties (Настройки конфигурации), выбрать Build (Создание приложения) и ввести имя файла XML в поле XML Documentation File, При каждом создании исполняемого файла будет производиться обновление файла документации XML.

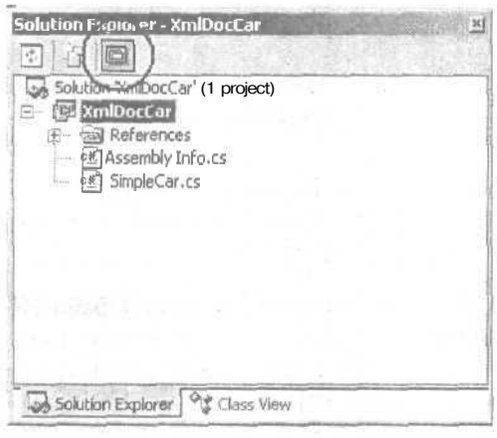


Рис. 5.12. Открытие диалогового окна Project Properties (Свойства проекта)

Просмотр файла документации в формате XML

Если мы откроем созданный нами файл simplecar.xml в интегрированной среде разработки Visual Studio.NET, появится окно, аналогичное представленному на рис. 5.13.

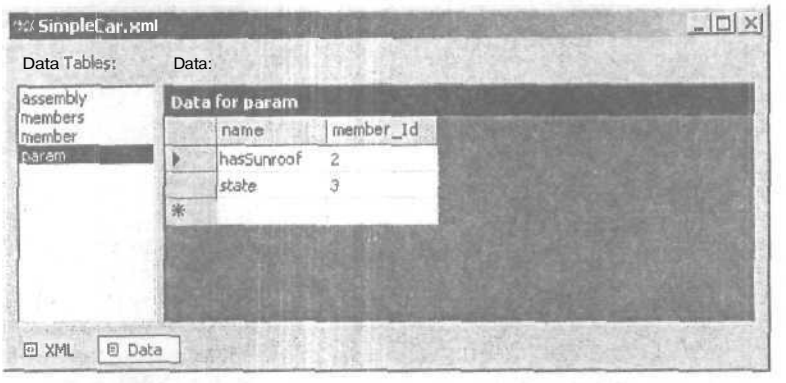


Рис. 5.13. Просмотр файла документации XML во внутреннем просмотрщике Visual Studio.NET

Если мы перейдем на вкладку XML, то нашему взору предстанет собственно код XML. Члены сборки помечены тегом <member>, поля помечены префиксом F, типы — T, а методы — M. Подборка символов форматирования, используемых в файлах документации в формате XML, представлена в табл. 5.4.

Таблица 5.4. Символы форматирования, используемые в документации в формате XML

Символ форматирования	Назначение
N	Пространство имен
T	Тип (класс, интерфейс, структура, перечисление или делегат)
F	Поле
P	Свойства типа (включая индексаторы)
M	Метод (включая конструкторы и перегруженные операторы)
E	Событие
!	Строка с информацией об ошибке. Компилятор C# генерирует такие сообщения об ошибке при обнаружении ссылок, которые не могут быть разрешены

Полученный файл XML можно преобразовать в файл HTML (это несложно сделать при помощи таблиц стилей XSL) или работать с ним программным образом при помощи типов .NET. XML — это формат с наибольшим количеством возможностей, но если нам по каким-то причинам это необходимо, мы можем сразу создавать файлы в формате HTML. О том, как это можно сделать, рассказывается ниже.

Создание документации в формате HTML

Задокументировать наш проект в формате HTML очень просто. Если нам необходимы дополнительные теги и комментарии, они добавляются в исходный код точно так же, как и теги XML (используется тот же самый синтаксис). После того как все комментарии добавлены, выберем в меню Tools (Сервис) пункт Build Comment Web Pages (Создать web-страницы комментариев). Далее нам предложат выбрать, создавать ли документацию для всего решения или только для отдельных проектов (рис. 5.14).

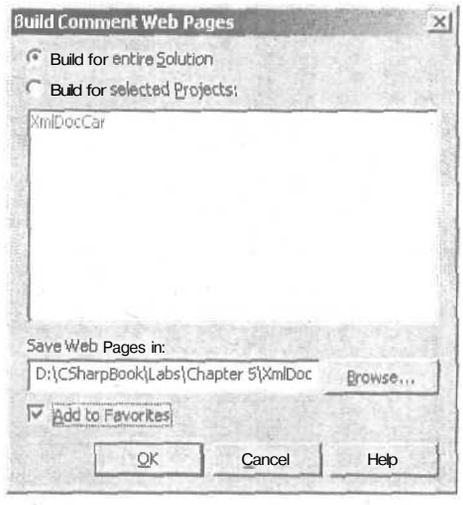


Рис. 5.14. Создание документации в формате HTML

В результате в каталоге нашего проекта появится новая папка с изображениями и файлами HTML, основанными на созданных нами комментариях. Пример файла документации в формате HTML представлен на рис. 5.15.

Код приложения XmlDocCar можно найти в подкаталоге Chapter 5.

Подведение итогов

Целью этой главы было знакомство с более сложными возможностями классов C# и интегрированной среды разработки Visual Studio.NET. В самом начале было рассмотрено создание пользовательского метода индексатора, при помощи которого можно обращаться к вложенным элементам объекта-контейнера точно так же, как и к элементам обычного массива. Далее мы рассмотрели возможности перегрузки операторов C#, при помощи которых пользовательские классы могут реагировать на стандартные операторы точно так же, как и встроенные типы данных.

Далее мы рассмотрели три способа, при помощи которых объекты могут принимать участие в двустороннем взаимодействии. Первые два подхода (применение делегатов и событий) — это официальные средства, рекомендуемые создателями C#.

Рис. 5.15. Электронная документация к проекту XmlDocCar

Третий подход — использование интерфейсов для обработки событий — это скорее специальное техническое решение, чем официально утвержденный протокол взаимодействия объектов. Однако и при помощи этого подхода мы также можем обеспечить безопасное для типов двустороннее взаимодействие объектов в C#.

Заканчивается глава описанием того, как средствами компилятора C# и среды разработки Visual Studio.NET можно создать электронную документацию для некоторого проекта.

Сборки, потоки и домены приложений 6

У всех приложений-примеров, которые мы создавали на протяжении первых пяти глав, была одна общая черта: все эти приложения были стандартными «автономными» приложениями, состоящими из единственного файла EXE. Однако, конечно же, это не единственный вариант создания приложений. Приложения C# и .NET в целом могут состоять из нескольких исполняемых файлов. Основные моменты, которые при этом стоит учитывать, связаны с возможностью повторного использования кода.

Как и COM, платформа .NET позволяет организовывать взаимодействие типов между двоичными файлами, созданными на разных языках. Однако межъязыковое взаимодействие в .NET развито гораздо больше, чем в COM. Например, .NET поддерживает межъязыковое наследование (представьте себе класс Visual Basic .NET, производный от C#). Для того чтобы познакомиться с организацией межъязыкового взаимодействия и преимуществами, получаемыми при создании приложений из нескольких двоичных файлов, нам придется рассмотреть логическое и физическое строение сборок .NET.

Помимо этого, в этой главе мы выясним различия между «частными» (private) сборками и сборками, предназначенными для общего доступа (shared), узнаем, как среда выполнения .NET определяет местонахождение сборки и для чего нужен Global Assembly Cache (GAC, глобальный кэш сборок). Мы также познакомимся с файлами конфигурации приложений — файлами в формате XML, которые содержат важную информацию для среды выполнения .NET (например, о том, какую версию сборки общего доступа следует выбирать).

Глава заканчивается рассмотрением вопросов, связанных с построением многопоточных приложений с использованием типов из пространства имен System.Threading. Если у вас есть опыт создания приложений Win32, то вы удивитесь тому, как просто и удобно реализовано в .NET управление потоками.

Проблемы с классическими двоичными файлами COM

Проблема повторного использования готового кода в двоичном формате стоит на повестке дня уже очень давно. К настоящему времени наиболее популярный способ решения этой проблемы — применение COM-серверов. Несмотря на то что создание и применение COM-серверов хорошо документировано и является фактически стандартом, каждый, кто сталкивался с этим на практике, знает, что совсем простыми эти вопросы назвать трудно. Разработчику приходится затрачивать очень много сил, чтобы создать для своего COM-сервера всю необходимую инфраструктуру (IDL, фабрики классов и т. п.) Если вы занимались COM, то, скорее всего, вам приходили в голову такие вопросы:

* Почему так трудно работать с разными версиями двоичного файла COM?
* Почему установка двоичного файла COM на компьютере пользователя сопряжена с такими сложностями?

Надо сказать, что в .NET все эти вопросы решаются гораздо проще. Причина заключается в том, что для двоичных файлов .NET используется новый формат, называемый сборкой (assembly). Однако прежде чем мы перейдем к рассмотрению архитектуры сборок, мы еще раз рассмотрим те проблемы, которые нам придется решать.

Проблема: работа с версиями COM

При работе с COM нам приходится создавать соклассы (coclasses) — пользовательские типы данных (то есть классы), реализующие интерфейсы COM (включая обязательный IUnknown). Затем сокласс упаковывается в двоичный файл DLL или EXE. После этого полученный двоичный файл развертывается на компьютере конечного пользователя и к нему могут обращаться другие программы.

Проблема, связанная с версиями COM, происходит из-за того, что в среде выполнения COM нет встроенного механизма, позволяющего гарантировать, что клиенту будет предоставлена нужная версия двоичного сервера COM. Конечно же, когда появляется новая версия сервера COM, мы ожидаем, что в ней будет реализована полная совместимость со старой версией. На практике же это происходит далеко не всегда.

Например, предположим, что на компьютере работает 10 приложений и при этом всем необходим MyCOMServer.dll версии 1.4. Вдруг на компьютере устанавливается еще одно приложение, а вместе с ним — MyCOMServer.dll версии 2.0. В результате вполне может получиться так, что все первые десять приложений разом перестанут работать. Такая ситуация не является такой уж редкостью, поскольку обеспечение полной обратной совместимости со старыми версиями — это исключительно сложная задача.

Ситуация, в которой необходимый программе модуль DLL может быть в любой момент перезаписан таким же модулем другой версии, получила название «ада DLL» (DLL Hell). Она характерна не только для COM-серверов, но и для самых обычных DLL, написанных на C. Как мы убедимся в этой главе, в мире .NET «ада DLL» больше нет. Платформа .NET исключает саму возможность возникновения

подобных ситуаций за счет реализации специальных технологий, таких как параллельное выполнение. Кроме того, в .NET реализована очень простая и надежная схема сосуществования разных версий двоичных файлов.

Проще говоря, .NET разрешает одновременное размещение на клиентском компьютере (и одновременное выполнение!) разных версий одного и того же двоичного файла. Таким образом, если клиент А обратится к MyDotNETServer.dll версии 1.4, а клиент В — к MyDotNETServer.dll версии 2.0, каждому из них будет предоставлена своя версия сервера. Привязка приложения к определенной версии двоичного файла может производиться при помощи файла конфигурации приложения.

Проблема: развертывание приложений COM

Взаимодействие со средой COM трудно назвать очень простой. Когда клиент COM обращается к соклассу, первый его шаг состоит в загрузке библиотеки COM путем вызова CoInitialize() для использования определенным потоком. Далее клиент производит дополнительные вызовы к среде выполнения COM (CoCreateInstance(), CoGetClassObject() и прочие) для загрузки этой библиотеки в память. В конечном итоге клиент COM получает ссылку на интерфейс, которую можно использовать для взаимодействия с соклассом.

Чтобы среда выполнения COM смогла обнаружить и загрузить двоичный файл COM-сервера, в первую очередь она должна знать, где физически этот файл лежит. Для этого COM-сервер должен быть правильно зарегистрирован. На первый взгляд регистрация не должна вызывать никаких сложностей: достаточно, чтобы программа установки или утилита, поставляемая с операционной системой, запустила необходимый код в двоичном файле COM-сервера (DllRegisterServer() в DLL или WinMain() в EXE). При этом COM-сервер сам внесет в реестр все необходимые записи: самого класса COM (CLSID), ингерфейса (IID), библиотеки (LIBID), приложения (AppID) и прочие.

Проблема заключается в том, что многочисленные записи в реестре, относящиеся к COM-серверу, и сам двоичный файл COM-сервера никак физически не связаны между собой. За счет этого их взаимосвязь является очень хрупкой: достаточно конечному пользователю переместить двоичный файл COM-сервера в другое место или просто переименовать его, как множество приложений могут перестать работать.

Если мы используем модель распределенного COM, при котором COM-сервер может быть расположен на удаленном компьютере в сети, возникают свои сложности. Например, если к COM-серверу обращаются 100 клиентских компьютеров в сети, а сам двоичный файл лежит на каком-то сетевом сервере, то необходимые записи в реестр должны быть внесены на 101 компьютере. Естественно, в реальной работе при этом не избежать головной боли.

Плагформа .NET упрощает развертывание приложений на клиентском компьютере за счет того, что записи о двоичных файлах .NET (сборках) вообще не вносятся в реестр. Вот так, просто и понятно. Вместо этого вся необходимая информация хранится в самих сборках. Все развертывание приложений .NET обычно сводится к копированию файлов на клиентский компьютер (или в общую папку на сервере). Прощай, HKEY_CLASSES_ROOT!

Обзор сборок .NET

Теперь, когда нам известны существующие проблемы, мы рассмотрим, как они решаются в мире .NET. Приложения .NET создаются путем объединения любого количества сборок. Сборка — это двоичный файл (DLL или EXE), который содержит в себе номер версии, метаданные (информацию о самом себе), а также типы (классы, интерфейсы, структуры и т. п.) и дополнительные ресурсы (изображения, таблицы строковых данных и прочее). Главное, что необходимо осознать, — внутренняя организация сборки .NET совершенно не похожа на двоичные файлы COM или обычных приложений Win32.

Например, встроенный в процесс сервер COM экспортирует четыре функции (`DllCanUnloadNow()`, `DllGetClassObject()`, `DllRegisterServer()` и `DllUnregisterServer()`), которые обеспечивают среде выполнения COM возможность обращаться к содержимому этого сервера. .NET DLL, с другой стороны, обязательно должна экпортировать только одну функцию — `DllMain()`.

Если COM-сервер реализован в виде файла EXE, то в этом файле реализован метод `WinMain()` как единственная точка входа в приложение. При запуске такой COM-сервер проверяет наличие самых разных параметров командной строки, которые и обеспечивают для этого COM-сервера такие же функциональные возможности, как и у COM DLL. В сборках .NET все по-другому. Несмотря на то что в качестве точки входа в приложение также используется метод `WinMain()` (или `Main()` для консольных приложений), вся остальная логика реализована совершенно иначе.

Одно из самых важных отличий сборок .NET от двоичных файлов серверов COM и любых других исполняемых файлов Win32 заключается в том, что сборки .NET содержат не платформенно-зависимые инструкции, а код на так называемом промежуточном языке Microsoft (Microsoft Intermediate Language, MSIL или просто IL). Этот язык не зависит ни от платформы, ни от типа центрального процессора. Код IL компилируется в платформенно-зависимые инструкции только во время выполнения. Таким образом, потенциально приложения .NET могут выполняться на платформах, отличных от Windows, на компьютерах с самыми разными центральными процессорами.

Помимо собственно инструкций на языке IL, каждая сборка .NET содержит в себе информацию о каждом типе сборки и каждом члене каждого типа. Эта информация генерируется полностью автоматически. Например, если вы создали на любом из .NET-совместимых языков класс Joystick, то соответствующий компилятор создаст метаданные, описывающие все поля, методы, свойства и события, которые определены в этом классе. Среда выполнения .NET использует метаданные для определений местонахождения типов (и их членов) внутри сборки, для создания экземпляров объектов и вызова удаленных методов.

В отличие от двоичных файлов классических серверов COM, любая сборка .NET содержит манифест — набор метаданных о самой сборке. Манифест содержит информацию о всех двоичных файлах, которые входят в состав данной сборки, номере версии сборки, а также, что очень важно, — сведения обо всех внешних сборках, на которые ссылается данная сборка (в отличие от классического сервера COM, в котором внешние зависимости не документировались). Таким образом, можно считать, что сборки .NET являются полностью самодокументируемыми.

Сборки из одного и нескольких файлов

Любая сборка может состоять из нескольких модулей. Модуль — это файл сборки .NET. В этом свете можно рассматривать сборку целиком как «единицу развертывания», состоящую из одной или нескольких частей (сборки иногда даже называют «логическими DLL»). Чаще всего сборка состоит из единственного файла. В этом случае между «логическим двоичным файлом» — сборкой и «физическим двоичным файлом» — модулем сборки существует отношение «один-к-одному». Вся информация сборки будет расположена в единственном физическом файле, как представлено на рис. 6.1.

Рис. 6.1. Сборка из одного файла

Основная цель, которая преследуется при создании сборки из нескольких файлов, — это более эффективная загрузка приложения. Предположим, что удаленный клиент обращается к многофайловой сборке, состоящей из трех файлов. Если для тех возможностей приложения, которыми пользуется клиент, необходим только один из трех файлов, среда выполнения .NET загрузит только его. Если размер каждого из файлов сборки составляет один мегабайт, вы почувствуете разницу.

Многофайловые сборки объединяются в единое целое манифестом, в котором хранится информация о всех модулях данной сборки. Манифест в принципе может быть расположен в отдельном файле, но гораздо чаще он помещается в один из модулей сборки — так, как это представлено на рис. 6.2.

Как, собственно, создаются многофайловые сборки — этот вопрос мы рассматривать не будем. Он хорошо освещен в электронной документации к C#. Создавать многофайловые сборки совсем несложно — практически все, что для этого требуется — указать параметр /addmodule в командной строке компилятора C#.

Логическое и физическое представление сборки

Для любой сборки .NET, как однофайловой, так и многофайловой, можно использовать два представления — физическое и логическое. При физическом представлении сборка — это набор файлов (модулей), которые содержат пользовательские типы и ресурсы (рис. 6.3).

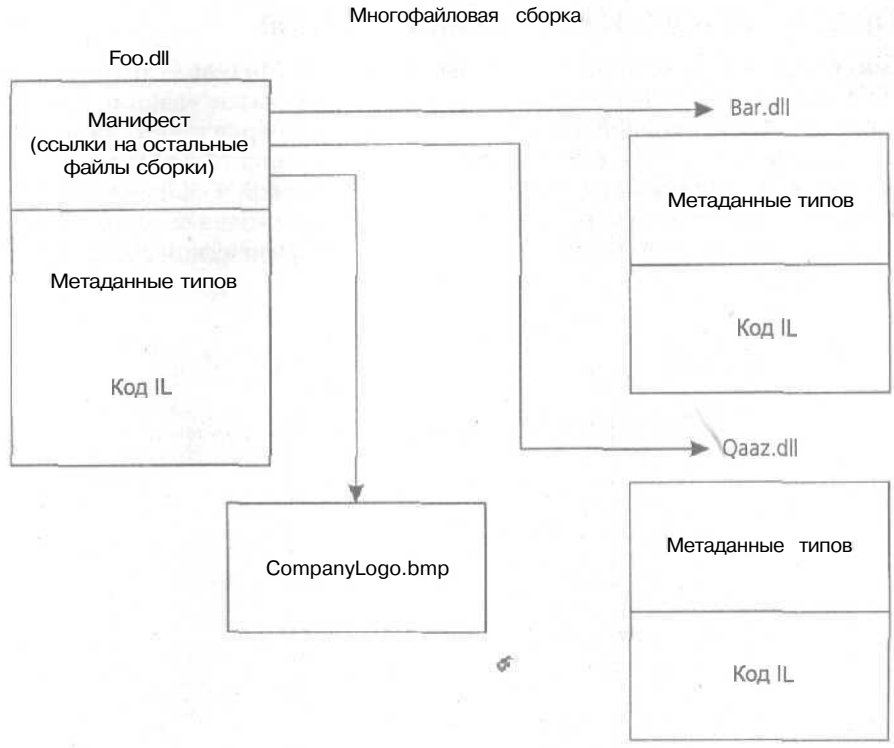


Рис. 6.2. Многофайловая сборка

Рис. 6.3. Физическое представление: сборка как набор модулей

Как правило, физическое представление сборки выбирает ее создатель. Пользователю же сборки гораздо важнее ее логическое представление, в котором сборка — это набор открытых типов, используемых в приложении («внутренние» типы — это, как правило, служебные типы, используемые другими типами той же самой сборки). Логическое представление сборки показано на рис. 6.4.

В качестве примера можно взять сборку System.Drawing.dll из библиотеки базовых классов .NET. На физическом уровне — это единственный файл dll, а в то же время на логическом уровне — это иерархия взаимосвязанных типов, которые используются для вывода изображений (в чем легко убедиться при помощи ILDasm.exe — рис. 6.5).

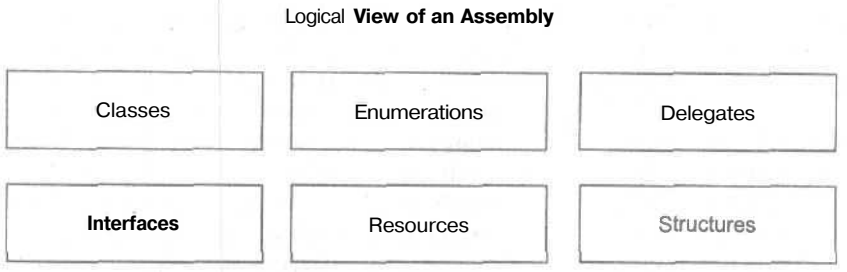


Рис. 6.4. Логическое представление; сборка как набор типов

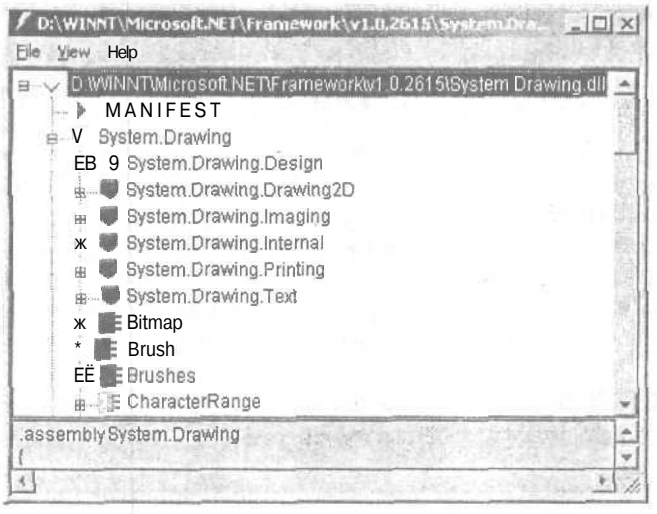


Рис. 6.5. Логическое представление сборки System.Drawing.dll

После того как мы рассмотрели разные представления сборок, мы кратко познакомимся с теми преимуществами, которые обеспечивают сборки .NET.

Сборки обеспечивают повторное использование кода

Повторное использование кода — это проблема, которая стоит перед разработчиками уже очень долго. Следует признать, что решение этой проблемы, предложенное разработчиками концепции сборок .NET, является лучшим из всего, что существует на настоящий момент.

Как мы уже говорили, типы и ресурсы, находящиеся внутри сборки, могут совместно использоваться самыми разными приложениями — точно так же, как типы и ресурсы двоичных файлов COM. Однако в отличие от COM в .NET вы можете не только разрешить такое повторное использование кода, но и явно запретить его! Это делается путем разделения сборок на открытые (те, которые можно использовать для общего доступа) и частные (private) — используемые только в рамках единственного приложения. По умолчанию все сборки помечаются как частные. Такая возможность позволяет нам гораздо проще развертывать приложения и работать с разными их версиями.

Как и в COM, в .NET повторное использование кода не ограничено рамками единственного языка программирования. Сборки .NET, которые будут успешно взаимодействовать друг с другом, можно создавать и на C#, и на Visual Basic.NET, и на любых других .NET-совместимых языках. Более того, в .NET предусмотрен специальный набор правил (Common Language Specification, CLS), следование которому позволит гарантировать, что любой модуль на любом .NET-совместимом языке будет нормально взаимодействовать с любым другим модулем на любом другом языке, даже несмотря на различия в возможностях разных языков.

Повторное использование кода, независимое от языка, в .NET реализовано гораздо лучше, чем в COM. Например, в классическом COM разработчики не могут производить тип в одном языке от типа из модуля, созданного на другом языке (выражаясь яснее, COM не поддерживает классического наследования). В .NET это вполне возможно.

Сборки — контейнеры для типов

Сборки используются как контейнеры для содержащихся в них типов и ресурсов, обеспечивая их уникальность и исключая возможность возникновения конфликтов имен. В .NET типы идентифицируются в том числе и по сборке, в которой они находятся. Таким образом, если в вашем приложении используются две сборки, в каждой из которых есть тип (класс, структура и т. п.) с одинаковыми названиями, конфликта имен не возникнет: среда выполнения .NET будет рассматривать эти два типа как полностью уникальные и независимые сущности.

В сборках предусмотрены встроенные средства самоописания и контроля версий

В мире COM вся ответственность за решение проблем, связанных с заменой старых версий компонентов новыми, лежала на программисте. Если новое приложение заменяло, к примеру, библиотеку MyComServer.dll версии 1.0 на версию 2.4 той же библиотеки, то вам оставалось только надеяться, что программист, создавший версию 2.4, позаботился о полной обратной совместимости с версией 1.0. Так, к сожалению, происходило не всегда, и иногда случалось, что при установке нового приложения старое переставало работать. Можно сказать, что проблема с версиями компонентов COM заключается в том, что ее решение возлагается полностью на программиста, а сама среда выполнения COM не предлагает для этого никаких средств.

Еще одна проблема, которую также можно отнести к проблеме версий, заключается в том, что в самоописании (при помощи IDL) двоичного сервера COM нет списка внешних зависимостей — модулей, от которых зависит нормальное функционирование данного сервера. Если какой-либо внешний модуль будет перемещен, переименован или удален (например, программой деинсталляции другого приложения), наш COM-сервер перестанет работать.

В .NET возникновение подобных проблем исключено — во-первых, за счет того, что на пользовательском компьютере одновременно могут сосуществовать разные версии одной и той же сборки, а во-вторых, поскольку в манифесте сборки явным образом указываются все внешние модули, необходимые для нормальной работы этой сборки.

У каждой сборки есть свой идентификатор версии (version identifier), который применяется ко всем типам и ресурсам внутри каждого модуля сборки. Используя информацию об этом идентификаторе, среда выполнения .NET гарантирует загрузку нужной версии сборки. Идентификатор версии сборки состоит из двух частей: дружественной текстовой строки (называемой информационной версией — informational version) и цифрового идентификатора (версией совместимости — compatibility version).

Предположим, что вы создали новую сборку с информационной версией MyInterestingString. Естественно, у каждой сборки должна быть и версия совместимости. Пусть в нашем случае она выглядит так: 1.0.70.3. Версия совместимости всегда состоит из четырех цифр, разделенных точками. Что же означают эти цифры?

Первая цифра — основной номер версии (major version), в нашем случае 1.

Вторая цифра — дополнительный номер версии (minor version), в нашем случае 0.

Третья цифра — номер сборки (build number), в нашем примере — 70.

Четвертая цифра — номер редакции (revision number), у нас — 3.

Ниже в этой главе мы рассмотрим, каким образом среда выполнения .NET использует номер версии сборки для загрузки именно тех двоичных файлов, которые нужны вызывающему их клиенту (речь идет о сборке общего пользования). Кроме того, мы также увидим, что поскольку в манифесте есть информация о всех внешних зависимостях для сборки, на основании нее среда выполнения .NET определяет и использует «последнюю известную рабочую конфигурацию» — набор сборок, которые нужны для правильной работы ассамблеи.

Сборки определяют контекст безопасности

Сборки .NET могут также содержать информацию о безопасности. В мире .NET основные параметры безопасности определяются именно на уровне сборки. Например, если сборка A хочет получить доступ к классу, расположенному внутри сборки B, то именно сборка B определяет, предоставлять такой доступ или нет. Ограничения системы безопасности, определенные в сборке, явно прописываются в ее манифесте. Мы не будем в этой книге подробно рассматривать систему безопасности .NET — отметим только, что доступ к содержимому сборки регулируется при помощи информации, находящейся в метаданных сборки.

Разные версии сборок могут выполняться параллельно

Возможно, одно из главных преимуществ сборок .NET заключается в том, что среда выполнения .NET может одновременно загружать и обеспечивать выполнение разных версий одной и той же сборки. Таким образом, на клиентском компьютере не только могут быть установлены разные версии сборок, но они еще и могут использоваться разными приложениями одновременно.

Определить нужную вашему приложению версию сборки можно при помощи конфигурационных файлов приложений. Эти текстовые файлы (точнее, файлы в формате XML) описывают нужную версию сборки, а также определяют местонахождение нужной приложению сборки. Мы научимся работать с файлами конфигурации приложений в этой главе.

Создание тестовой однофайловой сборки

Теперь, после знакомства с теорией, посвященной сборкам .NET, наша задача — создать библиотеку кода на C#. Физически эта библиотека кода будет представлять собой однофайловую сборку с именем CarLibrary. Чтобы приступить к созданию библиотеки, выберите в интегрированной среде разработки Visual Studio.NET новый проект Class Library (библиотека классов), как показано на рис. 6.6.

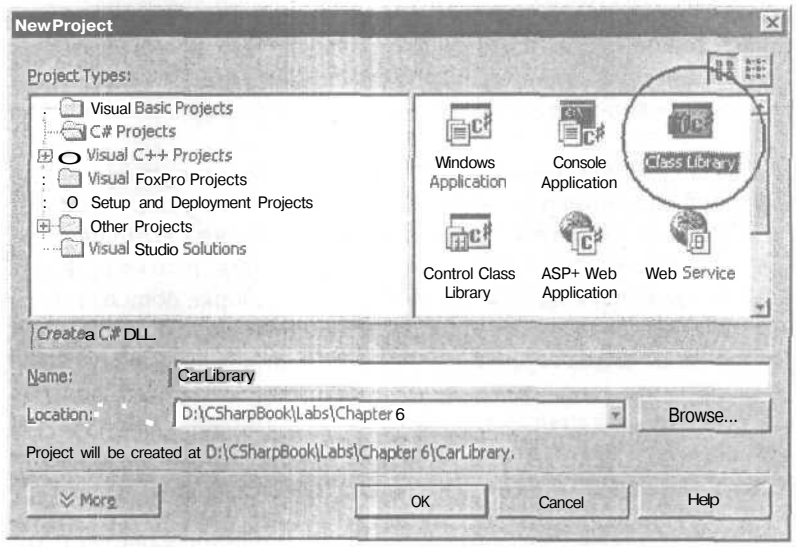


Рис. 6.6. Создание нового проекта Class Library (библиотека классов)

Первое, что мы сделаем — создадим абстрактный класс Car, в котором будет определен набор внутренних членов, определенных как protected. Доступ к этим внутренним членам будет осуществляться через свойства класса. Пусть в нашем классе будет единственный абстрактный класс с именем TurboBoost(). Кроме того, в классе будет использоваться перечисление EngineState. Изначальное определение нашего класса будет выглядеть следующим образом:

```
// Первая библиотека кода - CarLibrary.dll
namespace CarLibrary
{
using System;

public enum EngineState      // Для двух возможных состояний двигателя
{
      engineAlive,
      engineDead
}
public abstract class Car    // Абстрактный класс - базовый в нашей будущей иерархии
{
      // Защищенные данные о состоянии
      protected string petName;
      protected short currSpeed;
```

```
        protected short maxSpeed:
        protected  EngineState  egnState;

        public CarO {egnState * EngineState.engineAlive;}
        public Car(string name, short max, short curr)
        {
                egnState * EngineState.EngineAlive;
                petName = name; maxSpeed = max; currSpeed = curr:
        }

        public string PetName
        {
                get { return petName; }
                set { petName = value; }
        }

        public short CurrSpeed
        {
                get { return currSpeed; }
                set { currSpeed = value: }
        }

        public short MaxSpeed
        { get {return maxSpeed; } }

        public EngineState EngineState
        { get { return egnState; } }

        public abstract void TurboBoostO;
}
}
```

Теперь предположим, что у нас есть два класса, непосредственно производных от класса Car: MiniVan и SportsCar. Каждый из них реализует абстрактный метод TurboBoostO по-своему:

```
namespace  CarLibrary
t
using System;
using System.Windows.Forms; // Чтобы можно было использовать MessageBox

// Определение класса SportsCar
public class SportsCar : Car
{
        // Конструкторы
        public SportsCar(){}
        public SportsCarCstring name, short max. short curr) : base (name, max, curr) {]

        // Специфическая реализация метода TurboBoostO
        public override void TurboBoostO
        {
                MessageBox.Show("Ramming speed!", "Faster is better...");
        }
}

// Определение класса MiniVan
public class Minivan : Car
{
        // Конструкторы
```

```
        public MiniVan(){}
        public MiniVan(string name, short max, short curr) : base (name, max, curr){}

        // Реализация метода TurboBoost()
        {
                // Мини-вэны разгоняются неважно
                egnState - EngineState.engineDead;
                MessageBox.Show("Time to call AM". "Your car is dead");
        }
}
}
```

Оба варианта метода TurboBoost() выводят окно сообщения, используя при этом класс MessageBox из сборки System.Windows.Forms.dll. Чтобы наша сборка смогла использовать типы из сборки System.Windows.Forms.dll, проект CarLibrary должен включать ссылку на эту сборку. Ссылку можно добавить при помощи пункта Add Reference (Добавить ссылку) в меню Project (Проект) (рис. 6.7).

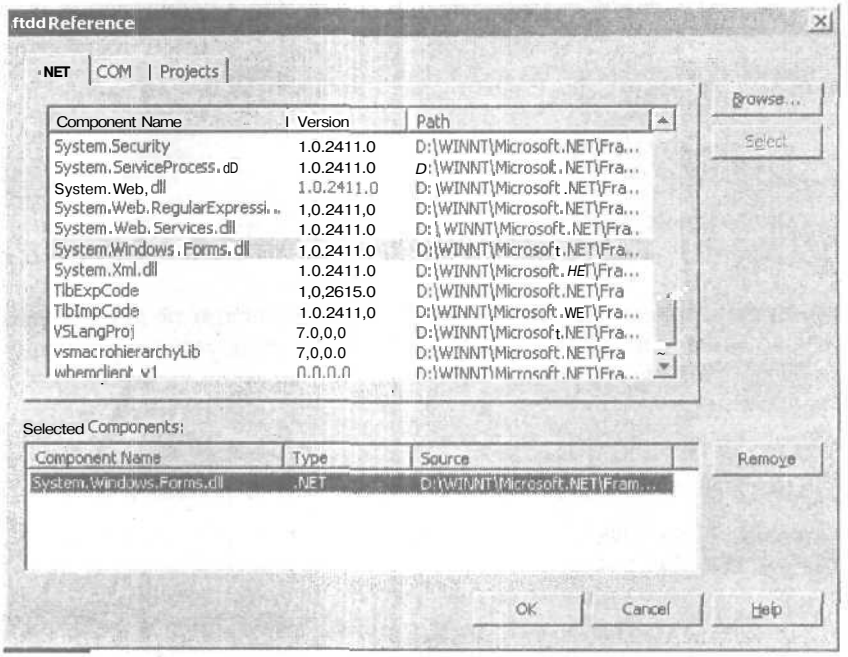


Рис. 6.7. Добавление ссылки на внешнюю сборку

Мы будем подробно разбирать пространство имен System.Windows.Forms и определенные в нем типы в главе 8. Как следует из имени этого пространства имен, типы, определенные в нем, относятся к элементам управления приложений Windows с графическим интерфейсом пользователя. Для целей этой главы пока достаточно сказать, что в этом пространстве имен определен используемый нами класс MessageBox — окно сообщения.

Последнее, что нам нужно сделать, — откомпилировать только что созданную нами библиотеку кода.

Клиентское приложение C#

Поскольку и класс SportsCar, и класс MiniVan объявлены как public, мы можем использовать эти классы в любых приложениях. Этим мы и займемся в ближайших разделах. Вначале мы создадим клиентское приложение C#, использующее эти классы, а затем — клиентское приложение с этими классами на Visual Basic.NET.

Приступим к созданию клиентского приложения C#, Первое, что нужно сделать — выбрать в качестве типа приложения C# Console Application (консольное приложение C#). Далее следует с помощью того же диалогового окна Add References (Добавить ссылку), представленного на рис. 6.7, включить в список ссылок для нашего приложения только что скомпилированную библиотеку CarLibrary.dll, выбрав ее при помощи кнопки Browse (Просмотр),

Добавив в наш проект ссылку на сборку CarLibrary.dll, мы тем самым гарантируем, что при первой же попытке запустить наше приложение интегрированная среда разработки Visual Studio.NET создаст полную копию CarLibrary.dll в каталоге Debug (рис. 6.8).

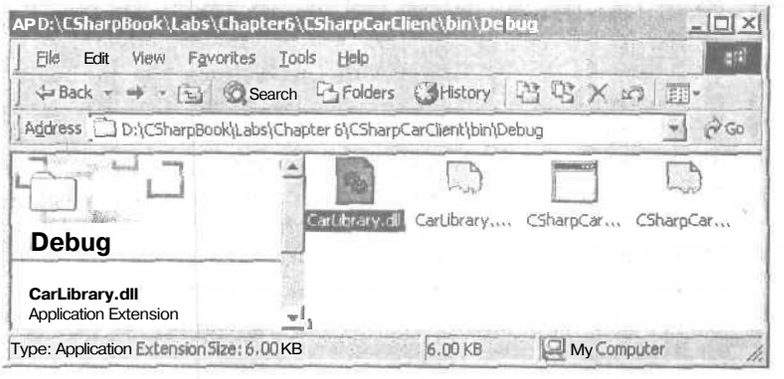


Рис. 6.8. Локальные копии сборок, на которые есть ссылки в приложении, помещаются в каталог Debug

Отличия от классического COM, в котором поиск используемых двоичных файлов производился через реестр, как мы видим, налицо.

После создания ссылки на сборку CarLibrary.dll мы можем использовать содержащиеся в ней типы. Наше клиентское приложение C# будет выглядеть следующим образом;

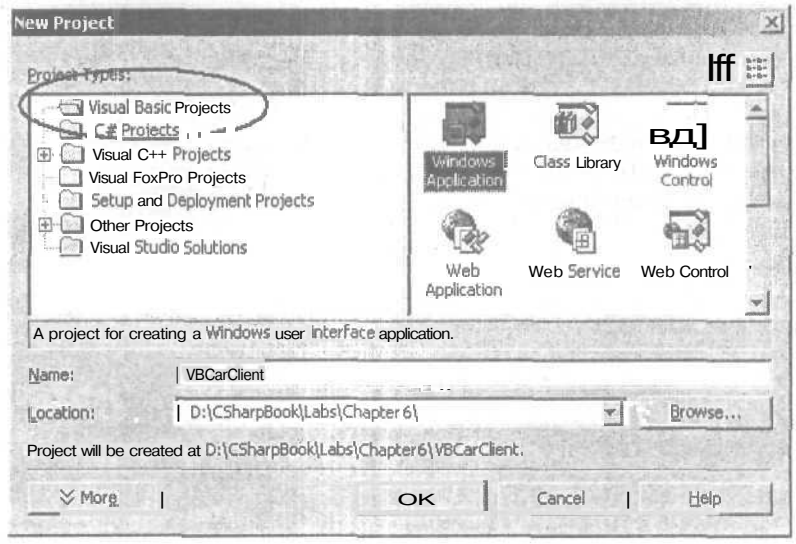
```
// Первый опыт использования собственной библиотеки кода
namespace CSharpCarClient
{
using System;

// Используем типы из CarLibrary
using CarLibrary;

public class CarClient
{
      public static int Main(string[] args)
      {
```

```
            // Создаем автонобиль спортивной модели
            SportsCar viper = new SportsCar("Viper", 240, 40);
            viper.TurboBoost();

            // Создаем мини-вэн
            MiniVan mv = new MiniVan();
            mv.TurboBoost();

            return 0;
        }
    }
}
```

Это приложение очень похоже на все те примеры, с которыми мы работали до этого. Единственное отличие заключается в том, что мы используем в нем специально заготовленную внешнюю библиотеку типов. При запуске программы все должно получиться так, как и ожидалось — то есть должны появиться два окна сообщения.

Клиентское приложение Visual Basic.NET

В состав Visual Studio.NET входят четыре языка, с помощью которых можно создавать код IL («управляемый код» — managed code): C#, Visual Basic.NET, JScript.NET и Managed C++ (MC++). Одна из многих хороших особенностей Visual Studio.NET — это то, что приложения на любом из этих языков создаются в одной и той же единой интегрированной среде разработки. Поэтому процесс создания клиентского приложения на Visual Basic не составит для нас никаких трудностей. Просто выберем в качестве шаблона для нового проекта Windows Application из контейнера Visual Basic Applications и дадим новому проекту название VBCarClient (рис. 6.9).



Рис, 6.9. Создание приложения VB.NET на основе шаблона Windows Application

Как и в Visual Basic 6.0, этот проект обеспечивает шаблон для создания приложений с главным окном и графическим интерфейсом пользователя. Однако между шаблонами VB.NET и VB 6.0 есть существенная разница: шаблон VB.NET построен на основе подкласса типа Form, который сильно отличается от объекта Form в VB 6.O.

После того как мы создали новый проект, следующая наша задача — добавить в этот проект ссылку на CarLibrary.dll. Это делается точно так же, как в клиентском приложении C# (см. предыдущий раздел). Далее можно приступать к созданию кода. Чтобы сократить названия объектов, мы начнем с того, что явно укажем все пространства имен, классы из которых будут использоваться в нашем приложении. В C# для этого использовалось ключевое слово using, а в VB.NET — imports. Мы должны открыть код для нашей формы VB.NET и записать следующие строки:

```
' Как и в C#, в VB.NET удобно явно указать пространства
' имен, классы из которых нам потребуются
Imports System
Imports Systemn.Collections
...
Imports CarLibrary
```

Следующая задача — создать графический интерфейс пользователя, который будет использоваться нашим приложением. Поместите на нашей форме два элемента управления Button (Кнопка), как показано на рис. 6.10.

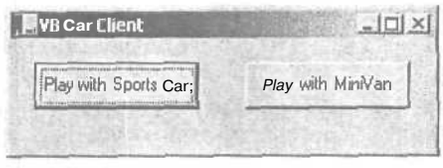


Рис. 6.10. Простой интерфейс для клиентского приложения VB.NET

Далее мы должны написать код, реагирующий на нажатие каждой кнопки (событие Click). Для этого просто щелкните дважды кнопкой мыши на каждой кнопке в окне IDE и впишите для каждой из кнопок соответствующий код. Для левой кнопки код будет выглядеть так:

```
Protected Sub btnCar_Click(ByVal sender As Object. ByVal e As System.EventArgs) Handles
btnCar.Click
        Dim sc As New SportsCar()
        sc.TurboBoost()
End Sub
```

А для правой — так:

```
Protected Sub btnMiniVan_Click(ByVal sender As Object. ByVal e As System.EventArgs)
Handles btnMiniVan.Click
        Dim sc As New MiniVan()
        sc.TurboBoost()
End Sub
```

Не подумайте, что скрытая цель этой книги — подготовка программистов на Visual Basic. Однако кое-что в этом коде может представить для нас интерес. Обратите внимание на то, как создаются объекты классов при помощи ключевого слова New. В отличие от VB 6.O классы в Visual Basic.NET теперь могут иметь настоя-

щие конструкторы! Таким образом, пустые скобки после имени класса, конечно, означают, что мы вызываем конструктор по умолчанию.

Откомпилируйте и запустите этот проект. При нажатии на каждую из кнопок каждый из наших автомобильных классов, созданных на C#, должен реагировать соответствующим образом — окном сообщения.

Межъязыковое наследование

Один из замечательных аспектов программирования в среде .NET — возможность использовать наследование классов между разными языками. Давайте создадим класс VB.NET, производный от нашего класса C# CarLibrary.SportsCar. Повторим еще раз для тех, кто почувствовал неладное: мы создаем класс VB.NET, производный от класса C#! В VB 6.0 это было невозможно, а в VB.NET — пожалуйста.

Наш производный класс VB.NET будет называться PerformanceCar. Для его создания выберите в меню Project (Проект) пункт Add Class (Добавить класс). Далее мы запишем код для создаваемого нами класса. Наследование в VB.NET производится при помощи ключевого слова Inherits. Класс PerformanceCar наследует методу C# с именем Car и замещает абстрактный метод TurboBoost(). Замещение в VB.NET производится при помощи ключевого слова Overrides.

```
' Да, теперь VB.NET поддерживает все основные принципы
' объектно-ориентированного программирования
Imports CarLibrary
Imports System.Windows.Forms

' Этот класс VB является производным от класса C# SportsCar
Public Class PerformanceCar
        Inherits CarLibrary.SportsCar

        ' Реализация абстрактного метода из класса Car
        Overrides Sub TurboBoost()
                MessageBox.Show("Blistering speed", "VB PerformanceCar says")
        End Sub
End Class
```

Если мы обновим нашу форму VB.NET, включив для нее дополнительную кнопку для PerformanceCar, то код для события Click этой кнопки может быть таким:

```
Protected Sub btnPreCar_Click(ByVal sender As Object, ByVal e As System.EventArgs)
Handles btnPerfCar.Click

        Dim pc As new PerformanceCar()
        pc.PetName = "Hank" ' Унаследованное свойство

        ' А какой класс является для нас базовый?
MessageBox.Show(pc.GetType().BaseType.ToString(), "Base class of Perf car")

        ' Собственная реализация метода TurboBoostO
        pc.TurboBoost()
End Sub
```

При нажатии на кнопку btnPreCar появится окно сообщения, подобное представленному на рис. 6.11. Обратите внимание на то, как можно определить базовый класс в процессе выполнения программным образом.

Рис. 6.11. Межъязыковое наследование в действии

Таким образом, в .NET любое приложение очень удобно разбивать на отдельные двоичные блоки, которые могут создаваться на любых .NET-совместимых языках. Любое приложение на любом .NET-совместимом языке может не только создавать объекты классов, определенных в сборке, созданной на другом языке, но и производить свои собственные классы от базовых классов в той же самой «чужой» сборке.

Код приложений CarLibrary, CSharpCarClient и VBCarClient можно найти в подкаталоге Chapter 6.

Подробности манифеста CarLibrary

После того как мы с таким успехом использовали CarLibrary в наших приложениях, настало время подробно ознакомиться с содержимым этой сборки, а именно — с ее манифестом. Как мы уже выяснили, манифест — это метаданные (данные о данных) для сборки в целом. Можно сказать, что манифест сборки — это Росетский камень .NET, поскольку «надписи» на нем могут прочесть любые приложения, вне зависимости от того, на каком .NET-совместимом языке они написаны. Что же содержит в себе манифест? Вот ответ:

- имя и идентификатор версии;
- список всех внутренних модулей сборки;
- список внешних сборок, необходимых для нормального выполнения (ссылок);
- информацию о естественном языке, используемом в сборке, — русском, английском, немецком и т. д.;
- «сильное имя» (strong name) — специальный вариант имени сборки, который используется для сборок, предназначенных для общего пользования;
- дополнительную (необязательную) информацию, связанную с безопасностью и хранением ресурсов внутри сборки (подробнее о форматах ресурсов .NET будет рассказано в главе 10).

Все компиляторы .NET-совместимых языков автоматически создают манифест в процессе компиляции. В следующей главе будет рассказано, каким образом можно дополнить автоматически сгенерированный таким образом манифест, используя средства работы с атрибутами. Однако настала пора убедиться в справедливости всего вышеизложенного на реальном примере нашей сборки CarLibrary.dll. Первое, что нужно сделать — открыть эту сборку в ILDasm.exe. Вашему взору должен предстать список доступной информации для всей сборки и для каждого из ее типов (рис. 6.12).

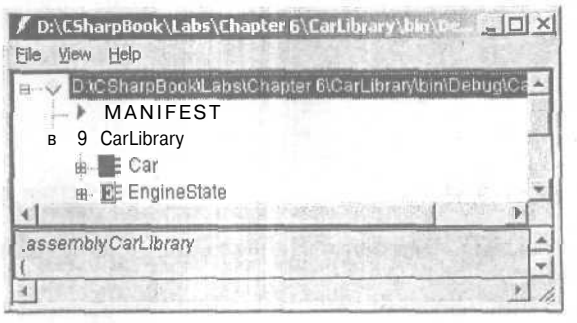


Рис. 6.12. Сборка CarLibrary в ILDasm.exe

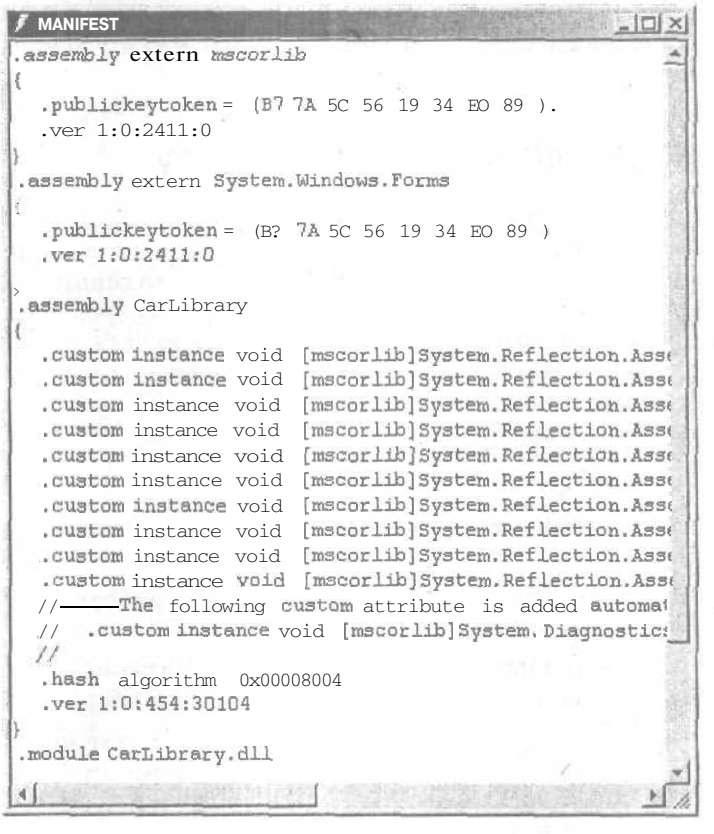


Рис. 6.13. Манифест сборки CarLibrary

Нас интересует манифест этой сборки. Чтобы открыть его, щелкните мышью на значке MANIFEST (рис. 6.13).

В первом блоке манифеста перечислены все внешние сборки, необходимые для нормальной работы CarLibrary.dll. Как мы видим, таких внешних сборок всего две: mscorlib.dll и System.Windows.Forms.dll. Все внешние сборки помечаются в манифесте тегом [.assembly extern]:

```
.assembly extern mscorlib
{
        .publickeytoken = (B7 7A 5C 56 19 34 EO 89)
        .ver 1:0:2411:0
}

.assembly extern System.Windows.Forms
{
        .publickeytoken = (B7 7A 5C 56 19 34 EO 89)
        .ver 1:0:2411:0
}
```

Как мы видим, в блоках [.assembly extern] присутствуют два выражения: [.publickeytoken] и [.ver]. Выражение [.publickeytoken] встречается только тогда, когда сборка определена для общего пользования (shared assembly). Это выражение используется для ссылки на «сильное имя» сборки (подробнее про этот механизм будет рассказано ниже). [.ver] — это, конечно же, цифровой идентификатор версии.

Кроме списка всех внешних ссылок, в манифесте также определен список всех внутренних модулей, из которых состоит сборка. Внутренние модули манифеста помечаются тегом [.module]. Поскольку CarLibrary — это однофайловая сборка, то этот тег встречается только один раз.

Для любой сборки можно также определить набор специальных атрибутов (подробнее о них — в главе 7). Эти атрибуты (в качестве примера можно привести атрибуты с информацией об имени компании, торговой марке и прочем) помечаются тегом [.custom]. Поскольку мы эти атрибуты не заполняли, их значения для CarLibrary.dll будут пустыми:

```
.assembly CarLibrary
{
        .custom instance void [mscorlib]
System.Reflection.AssemblyKeyNameAttribute::.ctor(string) = ( 01 00 00 00 00 )
        .custom instance void [mscorlib]
System.Reflection.AssemblyKeyFileAttribute::.ctor(string) = ( 01 00 00 00 00 )
        .custom instance void [mscorlib]
System.Reflection.AssemblyDelaySignAttribute::.ctor(bool) = C 01 00 00 00 00 )
        .custom instance void [mscorlib]
System.Reflection.AssemblyTrademarkAttribute::.ctor(string) = C 01 00 00 00 00 )
        .custom instance void [mscorlib]
System.Reflection.AssemblyCopyrightAttribute::.ctor(string) = ( 01 00 00 00 00 )
        .custom Instance void [mscorlib]
System.Reflection.AssemblyProductAttribute::.ctor(string) = ( 01 00 00 00 00 )
        .custom instance void [mscorlib]
System.Reflection.AssemblyCompanyAttribute::.ctor(string) = ( 01 00 00 00 00 )
        .custom instance void [mscorlib]
System.Reflection.AssemblyConfigurationAttribute::.ctor(string) = C 01 00 00 00 00 )
        .custom instance void [mscorlib]
System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = ( 01 00 00 00 00 J
        .custom instance void [mscorlib]
System.Reflection.AssemblytitleAttribute::.ctor(string) = ( 01 00 00 00 00 )

        .hash algontm 0x00008004
        .ver 1:0:454:30104
}
.module CarLibrary.dll
```

Дружественное имя нашей сборки (CarLibrary) помечено тегом [.assembly]. Тег [.ver] обозначает версию совместимости для сборки, а тег [.hash] — хэш, сгенери-

рованный для файла сборки. Обратите внимание, что для самой сборки CarLibrary тег [.publickeytoken] не используется, потому что CarLibrary — это частная (private) сборка, а не сборка для общего доступа (shared).

Все теги, используемые в манифесте сборки, представлены в табл. 6.1.

Таблица 6.1. Теги манифеста сборки

Тег	Назначение
.assembly	Помечает объявление сборки. Наличие этого тега означает, что этот файл — сборка .NET
.file	Помечает дополнительные теги в той же ассамблее
.class extern	Помечает классы, экспортируемые этой сборкой, но объявленные в других модулях
.exeloc	Помечает местонахождение исполняемого файла сборки
.mainfestres	Помечает внутренние ресурсы сборки (если таковые имеются). Мы подробнее познакомимся с этим тегом в главе 9
.module	Помечает объявление модуля. Используется для файлов модулей (двоичных файлов .NET без манифеста)
.module extern	Помечает внешние модули — модули данной сборки, на которые есть ссылки внутри текущего модуля
.assembly extern	Помечает внешние сборки — сборки, необходимые для нормальной работы данной сборки
.publickey	Содержит открытый ключ
.publickeytoken	Содержит маркер открытого ключа

Метаданные и код IL для типов CarLibrary

Как мы помним, главное, что у нас есть в любой сборке, — это типы. Любая сборка хранит в себе метаданные типов (сводную информацию для типов) и собственно код промежуточного языка IL для типов и их членов.

Мы познакомимся с форматом хранения метаданных и кода IL для типов на примере класса SportsCar. В этом классе определен метод TurboBoost(). Если мы щелкнем на строке с именем этого метода в окне ILDasm.exe, откроется новое окно с инструкциями IL для этого метода (рис. 6.14). Обратите внимание, что сам метод TurboBoostO помечен тегом [.method].

Внутренние данные класса, к которым возможен доступ из внешнего мира, помечаются тегом [.field] (поле). На рис. 6,15 представлен код IL для переменной currSpeed (familyозначает, что при объявлении этой переменной было использовано ключевое слово protected).

Свойства класса помечаются тегом [.property]. На рис. 6.16 представлен код IL для свойства CurrSpeed (как мы помним, это свойство обеспечивает доступ к переменной currSpeed). Конечно же, возможность использовать это свойство для чтения и (или) записи регулируется наличием или отсутствием соответствующих методов get_ и set_ (они помечены тегами [.get] и [.set]).

Чтобы просмотреть метаданные типов, достаточно в окне ILDasm.exe нажать Ctrl+M (рис. 6.17).

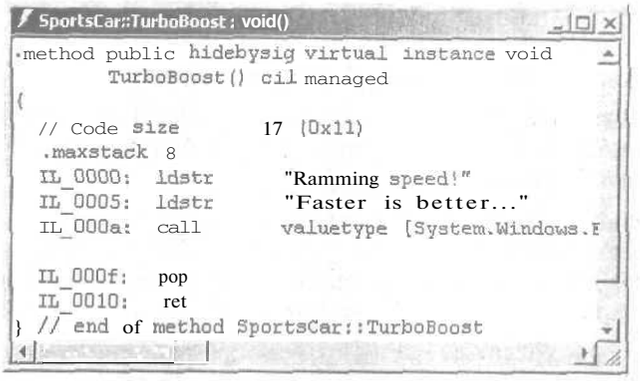


Рис. 6.14. Код IL для метода SportsCar.TurboBoost()

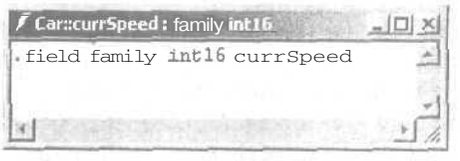


Рис. 6.15. Код IL для переменной currSpeed

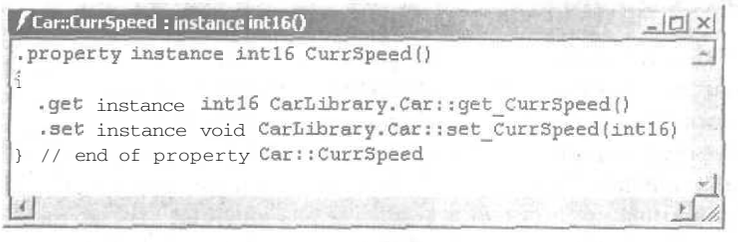


Рис. 6.16. Код IL для свойства CurrSpeed

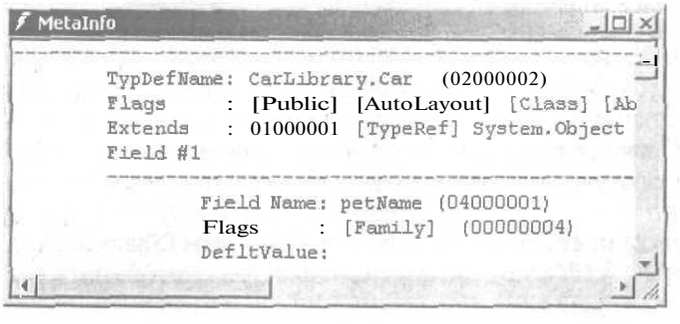


Рис. 6.17. Метаданные типов

Эти метаданные нужны среде выполнения .NET для создания объектов и вызова методов. Visual Studio.NET и другие средства используют метаданные ти-

пов во время разработки для проверки числа и типов параметров, передаваемых методам.

Подведем итоги того, о чем говорилось в предыдущих разделах;

- сборка — это набор модулей с самоописанием и идентификатором версии. Каждый модуль может содержать определения типов и (необязательно) дополнительные ресурсы;
- каждая сборка содержит метаданные, которые описывают все типы внутри нее. Среда выполнения .NET и средства разработки используют метаданные типов для поиска определений, создания объектов, проверки вызова методов, применения технологии IntelliSense и т. п.;
- каждая сборка содержит манифест, в котором перечислены все внутренние и внешние файлы, необходимые для нормальной работы сборки. Кроме того, в манифесте записана информация о версии, атрибутах сборки и прочие данные, характеризующие сборку.

Следующая часть настоящей главы посвящена различиям между частными сборками (private assemblies) и сборками, предназначенными для общего доступа (shared assemblies). Если вы пришли в мир .NET, обладая опытом работы с классическим COM, вы увидите, что реализация концепции совместно используемых компонентов претерпела в .NET существенные изменения.

Частные сборки

Любая сборка .NET может быть либо частной (private), либо сборкой для общего доступа (shared) — третьего не дано. У обеих версией сборок есть как общие черты, так и различия. Общихчерт гораздо больше: у всех сборок .NET одинаковая структура и схожее содержание (например, оба типа сборок предоставляют доступ к своим открытым членам). Различия между этими типами сборок состоят в особенностях их именования, политики версий и размещения сборок на компьютере пользователя. Знакомство с типами сборок мы начнем с первого из них — частных сборок, — как наиболее распространенного.

Частные сборки .NET — это наборы типов, которые могут быть использованы только теми приложениями, в состав которых они входят. Например, наша библиотека CarLibrary.dll — это частная библиотека, которая может быть использована приложениями CSharpCarClient и VBCarCUent (при создании этих приложений мы явно определили, что они будут использовать эту библиотеку, и добавили на нее ссылку).

Частные сборки должны находиться в основном каталоге приложения-владельца, они могут также находиться в подкаталогах этого каталога. Основной каталог приложения-владельца в .NET называется *каталогом приложения* (application directory). Вспомним, что когда мы добавили в проекты CSharpCarClient и VBCarCUent ссылки на сборку CarLibrary.dll, интегрированная среда разработки Visual Studio.NET автоматически создала в каталоге приложения (в момент разработки это — /Debug) копию CarLibrary.dll. Таким образом, размещение копии частной сборки там, где ее будут искать (в каталоге приложения), происходит автоматически.

Для того чтобы осознать, как это хорошо, можно вспомнить, что в классическом COM нам было необходимо создавать записи в разделе реестра HKEY_CLASSES_ROOT

и указывать пути к исполняемым файлам, используя InprocServer32 или LocalServer32. В .NET программа не потеряет работоспособности и в том случае, если вы переместите частную сборку с исполняемым файлом в другое место — главное, чтобы они были в одном каталоге. Для примера можно перетащить CSharpCarClient.exe и CarLibrary.dll на рабочий стол (рис. 6.18) — все будет работать.

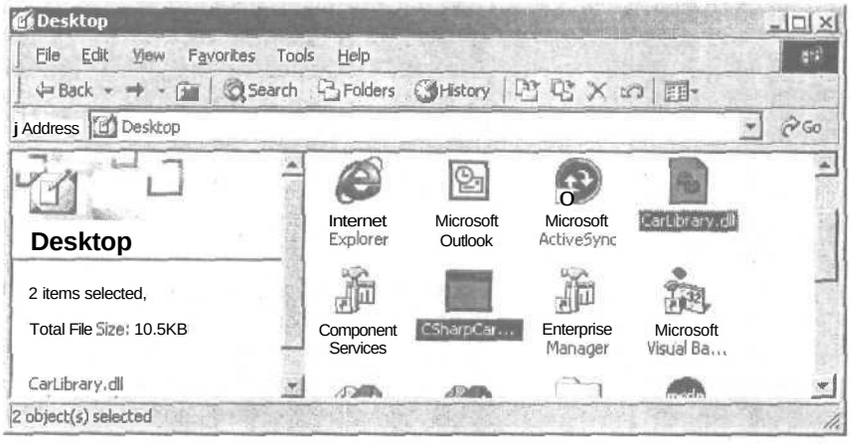


Рис. 6.18. Для установки приложения .NET достаточно просто скопировать файл и необходимые частные сборки на компьютер

Как, наверное, вы уже догадываетесь, простота установки означает и простоту удаления программы. Для этого достаточно просто удалить файлы этого приложения. Как тут не вспомнить классический COM, в котором при деинсталляции программы мы постоянно рискуем либо оставить в реестре «осиротевшие» записи, замусорив его, либо нарушить работоспособность какого-либо другого приложения.

Технология «зондирования»

Ниже мы довольно подробно опишем то, как среда выполнения .NET производит поиск сборок разных типов, Этот поиск в .NET обозначается специальным термином — «зондирование» (probing). Этот механизм и используемые для управления им средства могут быть весьма изощренными, однако для частных сборок первая часть всегда будет одной и той же. Предположим, в нашем приложении есть следующий код:

```
.assembly extern CarLibrary
{
...
}
```

Алгоритм поиска будет таким.

1. Среда выполнения .NET попытается обнаружить файл CarLibrary.dll в том же каталоге, в котором расположен исполняемый файл (в каталоге приложения).
2. Если CarLibrary.dU не обнаружен, го дальше среда выполнения попытается найти в том же каталоге файл CarLibrary.exe.

3. Если не найден и файл CarLibrary.exe, то среда выполнения .NET будет использовать для поиска другие способы, о которых речь пойдет ниже.

Идентификация частной сборки

Частная сборка идентифицируется по дружественному имени и числовой версии. И дружественное имя, и числовая версия записаны в манифесте сборки. Дружественное имя сборки — это имя двоичного модуля, который содержит манифест сборки. Например, в манифесте CarLibrary.dll можно найти следующие строки (номер версии не имеет значения):

```
.assembly CarLibrary as "CarLibrary"
{
...
        .ver  1:0:454:30104
}
```

я частной сборки среда выполнения .NET вообще не использует никакой по- в отношении версий. В этом просто нет необходимости: частная сборка находиться на своем месте в единственном экземпляре. Если несколько или одинаковых версий одной и той же частной сборки будут находиться в местах файловой системы компьютера (что вполне вероятно), то друг дру- . они мешать не будут.

Частные сборки и файлы конфигурации приложений

Если нам необходимо явным образом указать среде выполнения .NET, где разыскивать ту или иную сборку, в нашем распоряжении файлы конфигурации приложений. Это — обычные текстовые файлы в формате XML. Вся необходимая информация записывается в них в специальные теги XML. Файлы конфигурации должны иметь то же самое имя, что и приложение, к которому они относятся, и расширение *.config.

В файлах конфигурации .NET можно указать подкаталоги, в которых среда выполнения будет искать сборку. Конечно же, обычно такое решение используется для больших приложений, в которых сборок много и есть смысл упорядочить их по разным подкаталогам. Например, предположим, что мы разработали коммерческое приложение с именем MyRadApplication, в каталоге которого находятся подкаталоги \Images, \B1n, \SavedGames и \OtherCoolStuff. При помощи файлов конфигурации мы можем указать те каталоги, в которых среда выполнения .NET будет производить поиск частных сборок при запуске приложения.

В качестве примера создадим файл конфигурации для одного из наших приложений — CSharpCarClient. Для такого эксперимента необходимо перенести файл частной сборки CarLibrary.dll из каталога \Debug в какой-нибудь подкаталог внутри этого каталога. Поместим его в подкаталог \Foo\Bar так, как это показано на рис. 6.19,

Далее нужно создать новый файл конфигурации с именем CSharpCarClient.exe.config (для этого вполне подойдет самый обычный блокнот — notepad.exe) и сохранить его в том самом каталоге, в котором находится исполняемый файл CSharpCarClient.exe. Файл конфигурации должен начинаться с тега <Configurati on>. Между этим тегом и соответствующим ему закрывающим тегом </Configuration> необходимо размес-

тить тег <assemblyBinding>, внутри которого в атрибуте privatePath и указывается нужный подкаталог с частной сборкой (если вам необходимо указать несколько подкаталогов, то они перечисляются через точку с запятой). Весь файл конфигурации в нашем случае может выглядеть так:

```
<configuration>
      <runtime>
            <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
                  <probing privatePath="foo\bar"/>
            </assemblyBinding>
      </runtime>
</configuration>
```

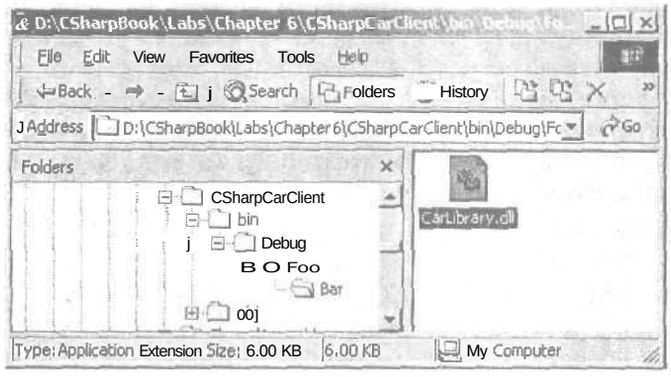


Рис. 6.19. Перемещаем файл частной сборки в подкаталог

Сохраним файл конфигурации и запустим приложение. Если все сделано правильно, то приложение сразу начнет работать.

Чтобы еще раз проверить, что будет в случае проблем с файлом конфигурации, просто переименуем его (например, так, как показано на рис. 6.20) и попробуем запустить приложение. Увы... Вывод прост: файл конфигурации приложения должен иметь то же самое имя, что и само приложение плюс расширение *.config.

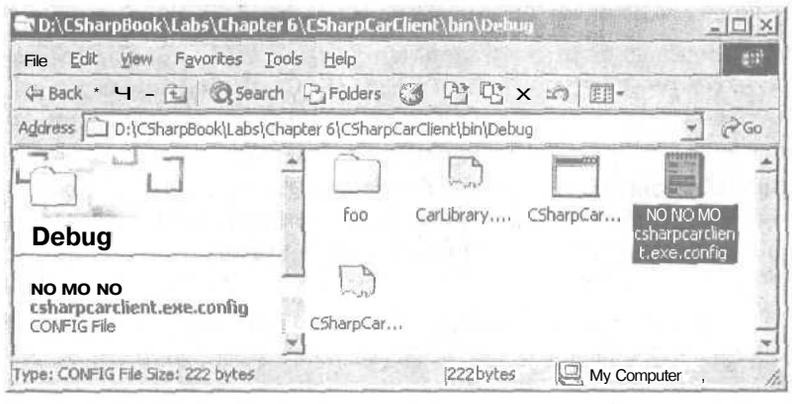


Рис. 6.20. Файл с расширением *.config должен иметь то же самое имя, что и само приложение

Процесс загрузки частной сборки в целом

Любой вызов на загрузку частной сборки может быть как явным, так и косвенным. Косвенный вызов происходит тогда, когда в тексте приложения встречается ссылка на эту сборку. В коде IL при этом такая внешняя сборка будет помечена тегом [.assembly extern]:

```
// Косвенный вызов на загрузку
.assembly extern CarLibrary
{
    ...
}
```

Явный вызов производится программным образом, при помощи метода System.Reflection.Assembly.Load(). Подробнее класс Assembly и его методы рассматриваются в главе 7, а для наших целей пока достаточно сказать, что этот метод позволяет указывать имя, версию, «сильное имя» и «культурную информацию» - данные о естественном языке, использованном в сборке. Впрочем, все эти параметры сразу приводить вовсе не обязательно. Вызов этого метода может выглядеть, к примеру, следующим образом:

```
// Явная загрузка сборки
Assembly asm = Asembly.Load("CarLibrary");
```

Все вместе — имя, версия, «сильное имя» и «культурная информация» называется ссылкой на сборку (assembly reference, AsmRef). Механизм, который отвечает за обнаружение сборки по AsmRef называется распознавателем сборок (assembly resolver), и он входит в структуры среды выполнения .NET.

Как уже говорилось, каталог приложения — это всего лишь каталог на жестком диске (например, C:\MyApp), в котором находятся все файлы приложения. Если есть необходимость, в каталоге приложения могут создаваться подкаталоги (например, C:\MyApp\Bin, C:\MyApp\Tools и т. п.). Как правило, подкаталоги создаются, чтобы упорядочить размещение файлов.

Когда поступает запрос на загрузку сборки, среда выполнения .NET передает AsmRef распознавателю сборок. Если распознаватель определяет, что AsmRef относится к частным сборкам (этот вывод делается при отсутствии «сильного имени» для этой сборки в манифесте), то он предпринимает следующие шаги:

1. Пытается обнаружить в каталоге приложения файл конфигурации приложения. В этом файле могут быть указаны дополнительные подкаталоги для поиска в них сборки, а также (как мы увидим дальше) явные указания для применения политики в отношении версии сборки.
2. Если файл конфигурации не обнаружен, распознаватель пытается обнаружить нужную сборку в текущем каталоге приложения. Если файл конфигурации существует, поиск производится и во всех указанных в этом файле подкаталогах.
3. Если сборка не может быть обнаружена в подкаталоге приложения (и в подкаталогах, определенных для поиска), поиск прерывается и генерируется исключение TypeLoadException.

Схема этого процесса представлена на рис. 6.21.

Рис. 6.21. Процесс поиска частной сборки

Сборки для общего доступа

Как и частные сборки, сборки для общего доступа — это набор типов и необязательных ресурсов внутри модулей — двоичных файлов сборки. Главное различие между типами сборок заключается в том, что частные сборки предназначены для использования одним приложением (в состав которого они и входят), а сборки для общего доступа — для использования неограниченным количеством приложений на клиентском компьютере.

Как правило, сборки для общего доступа устанавливаются не в каталог приложения, а в специальный каталог, называемый глобальным кэшем сборок (Global Assembly Cache, GAC). Этот каталог расположен в каталоге <имя_диска>:\WinNT\Assembly(рис.6.22).

Применение глобального кэша сборок — это еще одно существенное отличие .NET от классического COM. COM-серверы могли размещаться где угодно в файловой системе компьютера, главное — чтобы они были зарегистрированы в системном реестре. В .NET совместно используемые двоичные файлы должны находиться в точно определенном централизованном хранилище — GAC.

Еще одно различие между различными типами сборок заключается в том, что в сборках для общего доступа используется дополнительная информация о версии. Эта дополнительная информация называется «общим именем» (shared name) или «сильным именем» (strong name). Кроме того, для сборок этого типа среда выполнения не игнорирует информацию о версии, а активно использует ее.

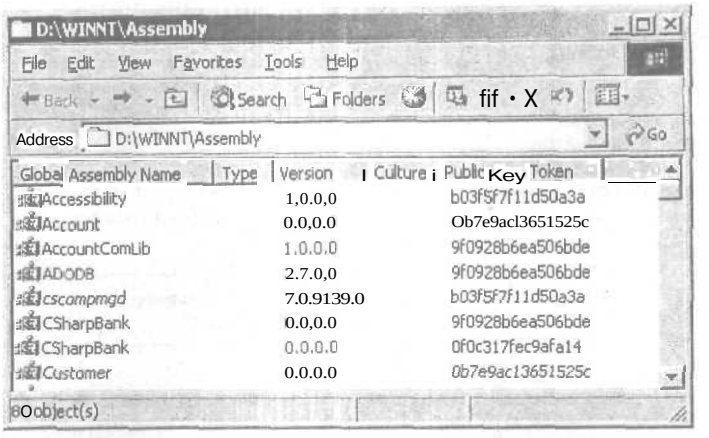


Рис. 6.22. Глобальный кэш сборок (GAC)

Проблемы с GAC?

Можете считать это заметкой на полях. При работе с Visual Studio.NET Beta 2 (которая была использована при написании этой книги) я неоднократно замечал, что некоторые компьютеры не могут правильно отобразить содержимое GAC. Проблема заключается в том, что для правильного показа GAC необходимо зарегистрировать расширение проводника Windows — сервер COM с именем shfusion.dll. Иногда во время установки Visual Studio.NET shfusion.dll по каким-то причинам не регистрируется. Чтобы решить проблемы с отображением GAC, достаточно просто зарегистрировать shfusion.dll при помощи системной утилиты regsvr32.exe.

Общие («сильные») имена сборок

При создании сборки для общего доступа нам обязательно потребуется создать уникальное общее имя. Само это имя состоит из следующей информации:

- дружественное текстовое имя и «культурная информация» (информация о естественном языке);
- идентификатор версии;
- пара открытый/закрытый ключ;
- цифровая подпись.

Создание общего имени основывается на криптографии открытого ключа. При создании сборки для общего доступа мы должны создать пару открытый/закрытый ключ (это делается очень просто). Эта пара будет использована .NET-совместимым компилятором, который затем поместит маркер открытого ключа в манифест сборки (пометив его тегом [.publickeytoken]). Закрытый ключ не помещается в манифест сборки. Вместо этого он используется для создания цифровой подписи, которая помещается в сборку. Сам закрытый ключ будет храниться в том модуле сборки, в котором находится манифест.

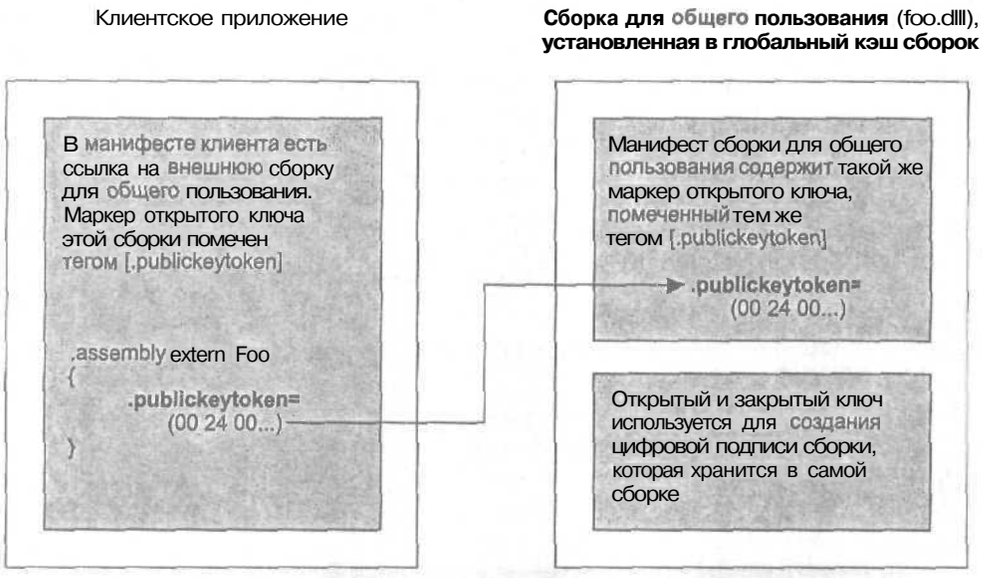


Рис. 6.23. Механизм проверки идентичности внешней сборки

Предположим, что клиент обратился к сборке для общего доступа (сам код для обращения к такой сборке ничем не отличается от кода для обращения к частной сборке). Еще при создании сборки общего доступа компилятор помещает ее открытый ключ в манифест. Во время выполнения среда выполнения .NET проверяет соответствие маркера открытого ключа сборки, запрашиваемой клиентом (как мы помним, эта информация для всех внешних общих сборок хранится в манифесте клиента), и маркера открытого ключа в самой сборке для общего пользования. Такая проверка гарантирует, что клиент получит именно ту сборку общего пользования, которая ему нужна. Весь этот механизм представлен в виде схемы на рис. 6.23.

Конечно же, применение пары открытый/закрытый ключ и цифровой подписи сборки не ограничивается теми моментами, которые были указаны выше. Однако мы на них останавливаться не будем. Интересующимся мы порекомендуем обратиться к электронной документации по .NET.

Создание сборки для общего пользования

Создание пары открытый/закрытый ключ для сборки производится при помощи утилиты sn.exe (от strong name — «сильное имя»). У этой утилиты есть множество параметров командной строки, однако пока нас интересует только один параметр: -k, который используется для создания набора ключей и сохранения их в указанном нами файле с расширением *.snk (рис. 6.24).

Если взглянуть иа содержимое созданного файла (в нашем случае theKey.snk), можно увидеть двоичный код пары ключей (рис, 6.25).

Что же делать дальше с этим файлом? Предположим, что мы создали новую библиотеку классов .NET, которая станет у нас сборкой для общего пользования, Называться она будет при этом SharedAssembly, а код ее будет таким:

```
using System;
using System.Windows.Forms;

namespace SharedAssembly
{
public class VWMiniVan
{
        public VWMiniVan(){}

        public void Play60sTunes()
        {
                MessageBox.Show("What a loooong, strange trip it's been...");
        }

        private bool isBustedByTheFuzz = false;
        public bool Busted
        {
                get { return isBustedByTheFuzz; }
                set { isBustedByTheFuzz = value; }
        }
}
}
```

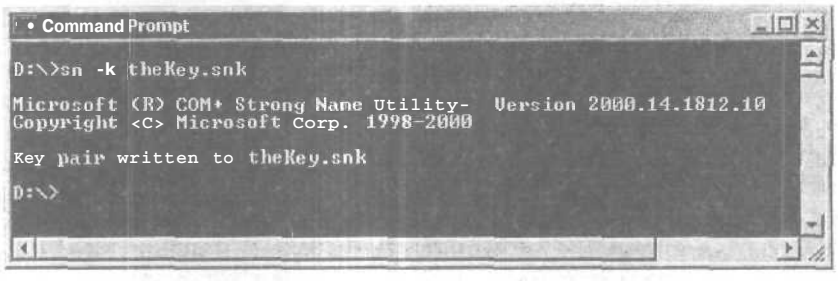


Рис. 6.24. Применение утилиты sn.exe

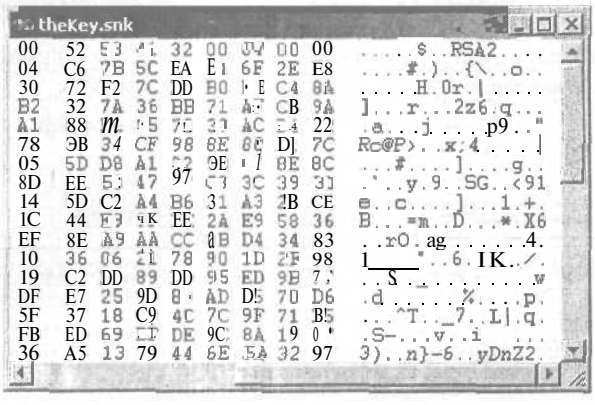


Рис. 6.25. Файл *.snk с парой ключей

Чтобы эта сборка стала общей, мы должны снабдить ее парой открытый/закрытый ключ из созданного нами файла theKey.snk. Это делается очень просто: откроем в Solution Explorer один из создаваемых автоматически файлов нашего

проекта, который называется AssemblyInfo.cs (рис. 6.26), и найдем в нем атрибут AssemblyKeyFile. Далее мы должны указать в качестве значения этого атрибута полный путь к созданному нами файлу *.snk:

```
[assembly: AssemblyKeyFile(@"D:\SharedAssembly\theKey.snk")]
```

Наша работа на этом заканчивается: все остальное сделает компилятор при создании сборки. Открытый ключ, который помещается в манифест сборки для общего пользования, можно просмотреть при помощи ILDasm.exe (рис. 6.27).

Код приложения SharedAssembly можно найти в подкаталоге Chapter 6.

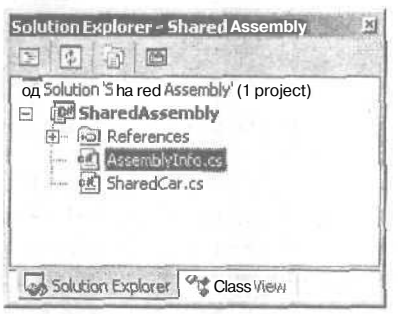


Рис. 6.26. Файл AssemblyInfo.cs

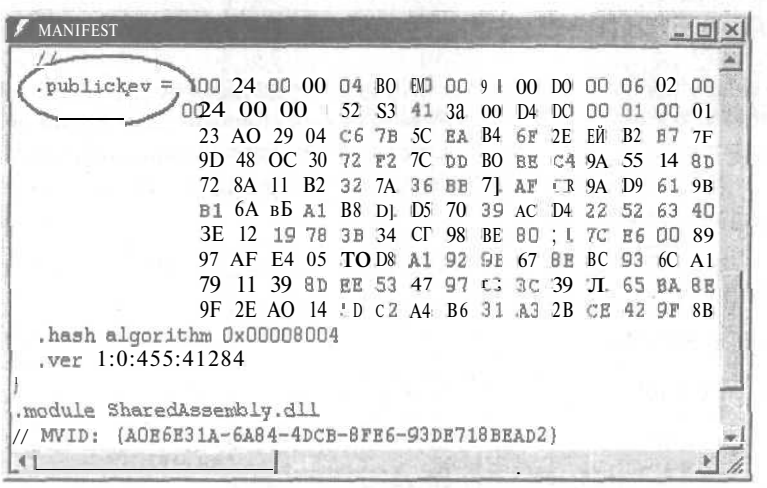


Рис. 6.27. Открытый ключ — отличительная особенность сборки для общего пользования

Установка сборки в глобальный кэш сборок (GAC)

После того как сборка для общего пользования готова, последнее, что осталось сделать — поместить ее в глобальный кэш сборок (Global Assembly Cache, GAC). Проще всего сделать это, просто перетащив мышью файл сборки в каталог <буква_диска>:\WinNT\Assembly (рис. 6.28). Другой вариант — воспользоваться утилитой gacutil.exe.

Для того чтобы устанавливать сборки в GAC или удалять их из него, мы должны обладать на этом компьютере правами администратора.

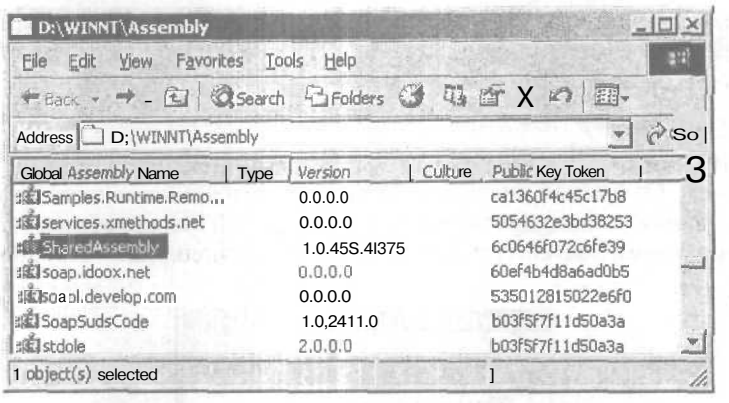


Рис. 6.28. Установка сборки в глобальный кэш сборок

После того как сборка для общего пользования установлена в GAC, она готова к использованию любыми приложениями. Удаление сборки из GAC производится предельно просто: нужно удалить нужный файл из каталога <буква_диска>:\WinNT\Assembly, например, при помощи правой кнопки мыши и *контекстного* меню.

Применение сборки для общего пользования в приложении

Теперь, когда у нас есть полностью готовая сборка для общего пользования SharedAssembly.dll, установленная в глобальный кэш сборок, самое время обратиться к ней из приложения. Для этого мы создадим новое консольное приложение C#, поместим в него ссылку на SharedAssembly.dll и запишем в нем следующий код:

```
namespace SharedLibUser
{
using System:
using SharedAssembly;

public class SharedAsmUser
{
        public static int Main(string[] args)
        {
                try
                {
                        VWMiniVan v = new VWMiniVan();
                        v.Play60sTunes();
                }
                catch(TypeLoadException e)
                {
                        // Не могу найти сборку!
                        Console.WriteLine(e.Message);
                }
                return 0;
        }
}
}
```

Вспомним, что когда мы создавали ссылку на частную сборку, интегрированная среда разработки Visual Studio.NET автоматически создавала локальную копию этой сборки в каталоге приложения. Если же мы создали ссылку на сборку, в которой есть открытый ключ (это — основной признак сборки общего пользования), такая локальная копия создаваться не будет. Среда выполнения .NET считает, что сборка с открытым ключом предназначена для общего пользования и искать ее следует не в каталоге приложения, а в глобальном кэше сборок.

Интегрированная среда разработки Visual Studio.NET позволяет также явным образом управлять созданием локальных копий сборок. Это делается из окна свойств для ссылки на сборку. Выберем нужную ссылку на сборку в окне Solution Explorer, откроем ее свойства и установим нужное значение в поле Copy Local (Копировать локально) — рис. 6.29.

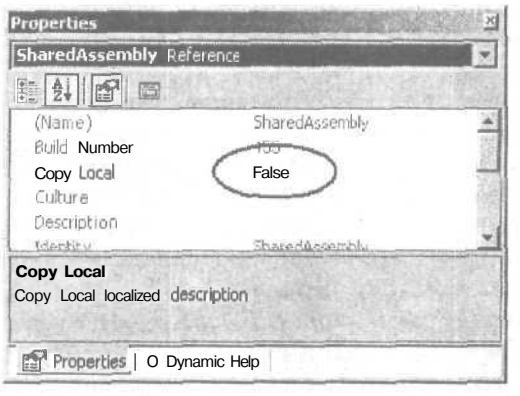


Рис. 6.29. Управление созданием локальных копий внешних сборок

Запустим созданное нами приложение. Все должно сработать без каких-либо проблем (рис. 6.30) — за счет того, что наше приложение использует сборку для общего пользования, помещенную нами в GAC.

Рис. 6.30. Приложение использует сборку для общего пользования из глобального кэша сборок

Код приложения SharedLibUser можно найти в подкаталоге Chapter 6. Перед запуском этого приложения убедитесь, что вы установили SharedLibrary.dll в GAC.

Политика версий .NET

Как нам известно, среда выполнения .NET не использует какой-либо политики версий в отношении частных сборок. Зато для сборок общего пользования номер версии имеет принципиальное значение. Учитывая его важность, вспомним, что

он состоит из четырех отдельных частей (например, 2.0.2.11). Для принятия средой выполнения .NET решения о совместимости версий главное значение имеют три первые части, как показано на рис. 6.31.

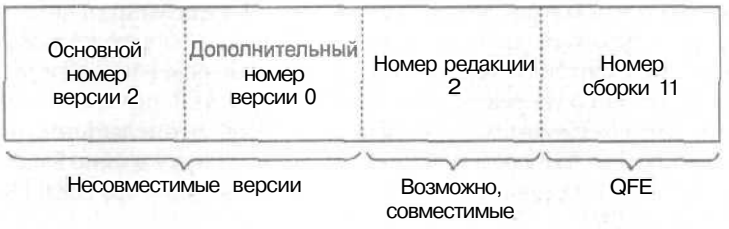


Рис. 6.31. Анатомия номера версии сборки

Если две сборки различаются по основному или дополнительному номеру версии, среда выполнения .NET считает их полностью несовместимыми. Подразумевается, что при изменениях в этих частях номера версии в самой сборке могут произойти такие изменения, как переименования методов, добавление новых типов или удаление старых, изменение наборов параметров и т. п. Поэтому если клиент обратится к сборке версии 2.0, а в глобальном кэше сборок будет присутствовать только версия 2.5, то клиенту будет сообщено о невозможности найти сборку (если нет специальных указаний в файле конфигурации этого клиента).

Если у двух сборок одинаковые основной и дополнительный номера версии, но разные номера редакций (например, 2.5.0.0 и 2.5.1.0), то среда выполнения считает, что они могут быть совместимыми (обратная совместимость вероятна, но не гарантируется). Номер редакции может измениться, например, при установке служебного пакета (Service Pack) для приложения.

Последняя часть номера версии часто называется также номером «быстрого исправления» (Quick Fix Engineering, QFE). Чаще всего этот номер меняется при установке программного исправления — патча. Среда выполнения .NET считает, что сборки, версии которых различаются только по номеру QFE, полностью совместимы между собой и в них совпадают все имена методов, наборы параметров поддерживаемые интерфейсы и т. п.

Запись информации о версии

Наверное, у вас уже созрел вопрос: а как можно присвоить сборке нужный номер версии? Как мы помним, среда разработки Visual Studio.NET автоматически добавляет в каждый проект файл AssemblyInfo.cs. В этом файле можно найти атрибут AssemblyVersion, которому изначально автоматически присваивается значение 1.0:

```
[assembly: AssemblyVersion("1.0.*")]
```

Любой проект C# начинается с версии 1.0. Если мы создаем новую версию сборки, то нам следует подумать об обновлении номера версии. Сама среда разработки .NET автоматически увеличивает только последние две части номера версии (которые в теге помечены значком *). Если вам нужно явным образом указать значе-

ния этих двух частей номера, просто запишите их в атрибут AssemblyVersion следующим образом:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Работа с разными версиями SharedAssembly

Чтобы окончательно прояснить все вопросы, мы поработаем с разными версиями сборки SharredAssembly.dll. Вначале мы внесем изменения в содержимое SharedAssembly.dll, сохраним новый вариант сборки с новым номером версии, а затем поместим этот новый вариант в GAC совместно с исходной версией. Далее мы будем обращаться из клиентского приложения к разным версиям этой сборки.

Первое, что мы сделаем, — внесем изменения в конструктор класса VWMiniVan для отображения версии уже существующей сборки:

```
public VWMiniVan()
{
    MessageBox.Show("Using version 1.0.0.0", "Shared Car");
}
```

Далее мы установим для атрибута AssemblyVersion значение 1.0.0.0 (так, как об этом рассказывалось в предыдущем разделе). После этого мы откомпилируем проект, создав новый файл SharedAssembly.dll.

Следующее, что мы должны сделать — удалить из глобального кэша сборок тот файл SharedAssembly.dll, который был туда установлен ранее. Удаление производится точно так же, как и удаление любого другого файла.

Затем переместим только что созданный нами файл сборки SharedAssembly.dll версии 1.0.0.0 в новую папку (назовем ее Version1), чтобы сохранить этот вариант сборки в неприкосновенности (рис. 6.32).

Далее установим SharedAssembly.dll версии 1.0.0.0 в GAC. Обратите внимание на номер версии, показываемый GAC для этой сборки (рис. 6.33).

Щелкнем правой кнопкой мыши на сборке в GAC и откроем ее свойства: мы увидим, что в качестве пути к этой сборке указан подкаталог Version 1. Далее перекомпилируем и запустим приложение SharedAssemblyUser. Оно должно запуститься без каких-либо проблем.

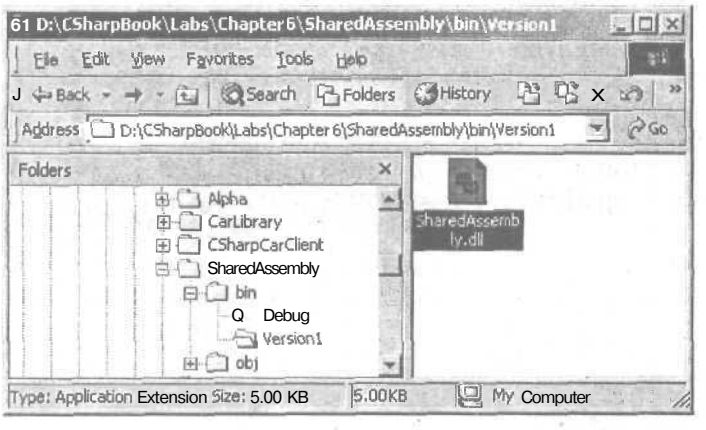
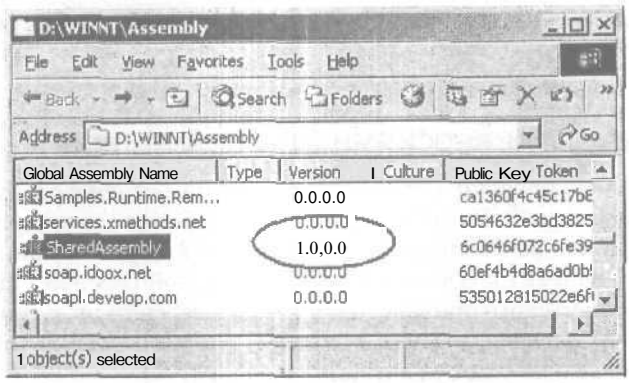


Рис. 6.32. Сохраняем SharedAssembly.dll версии 1.0.0.0 в отдельной папке



Рис, б.33. Помещаем сборку версии 1.0.0.0 в GAC

Создаем SharedAssembly версии 2.0

Чтобы проиллюстрировать политику версий .NET, мы внесем изменения в наш проект SharedAssembly. Добавим в класс VWMiniVan еще один член (который будет использовать специально созданное нами перечисление), для того чтобы у пользователя появился больший выбор мелодий. Кроме того, мы изменим текст сообщения, которое будет выдавать конструктор.

```
// Какую музыку вы предпочитаете?
public enum BandName
{
      TonesOnTail, SkinnyPuppy, deftones, PTP
}

public class VWMiniVan
{
      public VWMiniVan()
      { MessageBox.Show("Using version 2.0.0.0!", "Shared Car"); }
...
      public void CrankGoodTunes(BandName band)
      {
            switch(band)
            {
                  case BandName.deftones:
                        MessageBox.Show("So forget about me...");
                        break;
                  case BandName.PTP:
                        MessageBox.Show("Tick tick tock...");
                        break;
                  case BandName.SkinnyPuppy:
                        MessageBox.Show("Water vapor, to air...");
                        break;
                  case BandName.TonesOnTail:
                        MessageBox.Show("Oooooh the rain. Oh the raiin");
                        break;
                  default:
                        break;
            }
      }
}
```

Перед компиляцией изменим версию сборки на 2.0.0.0:

```
// Изменение в файле AssemblyInfo.cs...
[assembly: AssemblyVersion("2.0.0.0")]
```

После компиляции в каталоге Debug будет создан новый файл SharedAssembly.dll версии 2.0.0.0 (в то время как старый вариант этого файла версии 1.0.0.0 остался в неприкосновенности в подкаталоге Version1). Установим новый вариант SharedAssembly.dll в GAC. Теперь в GAC у нас мирно сосуществуют *две разные версии* одной и той же сборки (рис. 6.34).

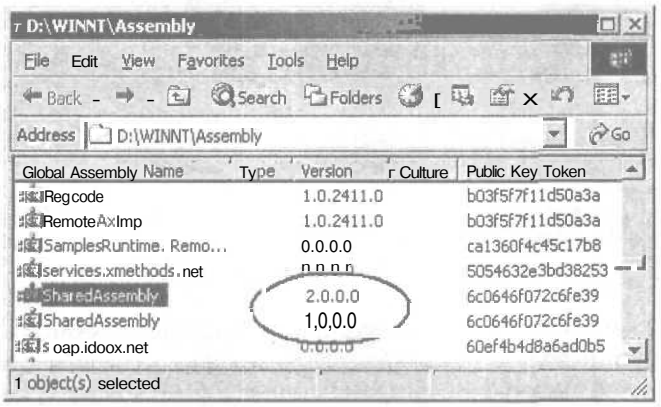


Рис. 6.34. В GAC могут одновременно находиться две версии одной и той же сборки

Теперь, когда в нашем распоряжении есть две версии одной сборки, нам предстоит узнать, как можно управлять загрузкой разных версий сборок при помощи файлов конфигурации приложений. Однако вначале мы опишем политику версии .NET по умолчанию — которая используется тогда, когда нет дополнительных указаний в файлах конфигурации приложений.

Политика версий .NET по умолчанию

Как мы уже говорили, если запрашиваемые клиентом и реально имеющиеся в GAC сборки отличаются друг от друга номером основной или дополнительной версий, то среда выполнения .NET посчитает их несовместимыми и клиенту будет выдано сообщение об ошибке. Если же различия в номерах версий — только на уровне номера редакции или номера сборки (QFE), то такие сборки .NET считает совместимыми. Однако что будет, если в GAC есть сборки версий 1.0.0.0 и 1,0.2.2, а клиент явно запрашивает версию 1.0.0.0? Ответ прост: среда выполнения .NET по умолчанию предоставит ему более свежий совместимый вариант сборки (то есть 1.0.2.2). Таким образом, гарантируется, что клиент получает не только совместимый вариант сборки, но вариант с наибольшим количеством исправленных ошибок.

Управление загрузкой разных версий сборок

Явным образом управлять загрузкой требуемой версии сборки (например, чтобы загружалась версия с меньшим номером QFE) можно при помощи файла конфигура-

ции приложения. Мы уже знакомы с ними — это файлы в формате XML, которые имеют то же самое имя, что и приложение (плюс расширение *.config), и находятся в каталоге приложения. Ранее мы уже рассматривали тегprivatePath, используемый для указания подкаталогов, в которых среда выполнения должна провести поиск сборок. Для того чтобы явно указать требуемую версию сборки, в файлах конфигурации используются тега <dependentAssembly> и <bindingRedirect>. Например, такой код заставляет среду выполнения .NET принудительно загружать сборку версии 2.0.0.0:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
          <dependentAssembly>
             <assemblyIdentity name="sharedassembly"
                  publicKeyToken="6c0646f072c6fe39"
                  culture=""/>

             <bindingRedirect oldVersion="1.0.0.0"
                              newVersion="2.0.0.0"/>

          </dependentAssembly>
       </assemblyBinding>
    </runtime>
</configuration>
```

Тег oldVersion используется для указания версии сборки, вместо которой мы будем принудительно использовать сборку другой версии — указанной в теге newVersion.

Проверим это на примере. Вначале мы создадим файл конфигурации приложения точно так, как представлено в тексте, и сохраним его в каталоге приложения (убедимся, что мы правильно присвоили имя этому файлу). Затем запустим программу. Должно открыться окно сообщения, представленное на рис. 6.35.

Рис. 6.35. Принудительно используем сборку версии 2.0.0.0

Если же мы изменим в файле конфигурации приложения значение атрибута newVersion на 1.0.0.0, то откроется другое окно сообщения (рис. 6.36).

Рис. 6.36. Теперь в файле конфигурации приложения указана версия 1.0.0.0

Таким образом, учитывая, что в глобальном кэше сборок могут храниться разные версии сборок, мы можем без проблем указывать нашему приложению ту версию сборки, которая должна быть загружена.

Администраторский файл конфигурации

У всех файлов конфигурации приложений, которые мы рассматривали в этой главе, есть одна общая особенность — каждый из этих файлов относился только к одному приложению. Однако платформа .NET позволяет использовать файл конфигурации, единый для компьютера и всех установленных на нем приложений. Этот файл носит имя machine.config и называется администраторским файлом конфигурации (administrator configuration file). Применение этого файла может представлять большое удобство, поскольку нет необходимости отдельно указывать политики версий и другие особенности запуска наших приложений для каждого из них в отдельности — все это можно сделать централизованно. Советуем вам найти этот файл на своем компьютере и познакомиться с его содержимым — вы найдете в нем дополнительные теги, которые также можно использовать в приложениях.

Теперь, когда мы обладаем достаточными сведениями о содержимом сборок и приемах работы с ними, мы рассмотрим тему, тесно связанную с предыдущей, — как работать с потоками и многопоточными приложениями .NET.

Работа с потоками в традиционных приложениях Win32

В зависимости от вашего прежнего опыта вы можете быть в большей или меньшей степени знакомы с созданием многопоточных приложений в традиционной среде Win32 или совсем не знать, что такое потоки. Поэтому мы начнем эту главу с рассмотрения основных принципов многопоточности и с обзора того, как многопоточность реализовывалась в приложениях Win32, а затем уже перейдем к созданию многопоточных приложений .NET.

Каждому традиционному приложению Win32 соответствует один (обычно) или несколько *процессов* (process). Процесс — это единица, которая характеризуется собственным набором внешних ресурсов (таких как COM-сервер) и выделенной приложению областью оперативной памяти. Для каждого файла EXE операционная система создает отдельную изолированную область в оперативной памяти, которой процесс пользуется в течение всего своего жизненного цикла.

Каждому процессу соответствует один (по крайней мере) или несколько потоков. Поток (thread) можно представить как специфический путь выполнения внутри процесса Win32 (thread в буквальном переводе с английского означает «нить»). Первый поток, создаваемый в процессе, называется первичным процессом (primary thread). В любом процессе существует по крайней мере один поток, который выполняет роль точки входа для приложения. В традиционных графических приложениях Windows такой точкой входа является метод `WinMain()`, а в консольных приложениях — метод `main()`.

Основная цель, для которой создаются многопоточные приложения (вместо использования единственного потока), — повышение производительности и сокращения времени отклика приложения. К примеру, однопоточные приложения могут раздражать конечного пользователя тем, что они не будут реагировать на его действия в течение длительного времени при выполнении какой-либо сложной операции (например, при печати длинного файла, выполнения большого вычисления или подключения к удаленному серверу). С другой стороны, однопоточные приложения обладают своими преимуществами — например, если данные изменяет только один поток, то

гораздо проще обеспечить целостность этих данных. Можно сказать, что однопоточные приложения автоматически являются «потокобезопасными» (thread-safe).

В Win32 главный поток может порождать дополнительные потоки, используя специальные функции Win32 АИ (например, `CreateThread()`). Каждый поток, как первичный, так и вторичный, становится частью пути выполнения процесса и получает возможность доступа ко всем данным в этом процессе.

Многопоточные приложения дают большой выигрыш в производительности на компьютерах с несколькими центральными процессорами. Однако применение таких приложений часто бывает полезным и на однопроцессорных системах, даже несмотря на то, что процессор в конкретный момент времени может заниматься обработкой только одного потока. Дело в том, что многопоточные приложения на однопроцессорном компьютере создают иллюзию одновременного выполнения сразу нескольких дел. Например, мы можем заставить фоновый рабочий поток выполнять какую-либо трудоемкую и длительную операцию — типа распечатки большого текстового файла. Однопоточное приложение занималось бы этим очень долго, а при использовании нескольких потоков другой поток вполне сможет одновременно (точнее, создавая иллюзию одновременности) обрабатывать ввод нового текста со стороны пользователя. Таким образом может быть достигнут значительный выигрыш в производительности. Но слишком много потоков в приложении — это тоже плохо. Центральному процессору придется затрачивать слишком много времени для переключения между потоками, и общая производительность работы приложения снизится.

Управление потоками производится на уровне операционной системы. Компьютеры с единственным центральным процессором в действительности не могут одновременно обрабатывать более одного потока. Иллюзия многозадачности достигается тем, что каждому потоку выдаются (в соответствии с его приоритетом) специальные кванты времени (time-slice). При исчерпании потоком выделенного ему кванта времени ресурсы центрального процессора передаются другим потокам, а первый поток вновь ждет своей очереди. Для того чтобы поток мог продолжить выполнение с того места, на котором его работа была остановлена, он обеспечивается возможностью записи в локальную память потока (Thread Local Storage, TLS) и отдельным стеком вызовов (рис. 6.37).

Рис. 6.37. Процесс традиционного приложения Win32

Проблемы одновременности и синхронизации потоков

Помимо того что переключение между потоками требует затрат времени центрального процессора, использование нескольких потоков может породить дополнительные проблемы. Например, предположим, один поток получил доступ к общим дан-

ным и начал вносить в них изменения. Далее его выполнение было отложено, и доступ на чтение к этим данным получил другой поток. Вполне может получиться так, что второй поток считает наполовину измененные данные — то есть данные, находящиеся в нецелостном состоянии. Если же оба потока будут вносить изменения в данные, то эти данные могут быть полностью испорчены.

Для того чтобы предотвратить возникновение подобных ситуаций, программисты Win32 используют специальные примитивы для работы с потоками, такие как критические секции, мьютексы и семафоры. Тем не менее многопоточные приложения остаются более подвержены сбоям, чем однопоточные. И конечно же, многопоточные приложения создавать гораздо сложнее.

Нельзя сказать, чтобы в .NET все эти проблемы развеялись как дым. Однако процесс создания многопоточных приложений в .NET радикально упрощен, а их надежность значительно повышена. Сделано это в основном за счет применения типов, определенных в пространстве имен System.Threading. Однако в некоторых специальных ситуациях, например для блокировки каких-либо общих данных, нам могут потребоваться дополнительные типы из других пространств имен.

Что такое домен приложения

Перед тем как приступать к рассмотрению типов пространства имен System.Threading, мы должны познакомиться с понятием доменов приложений (application domains, AppDomains). Как мы помним, приложение Win32 может состоять из одного или нескольких процессов, каждый из которых, в свою очередь, может порождать один или несколько потоков. В .NET в этой схеме появляется дополнительное звено: каждый процесс приложения .NET может состоять из одного или нескольких доменов приложений, а в рамках домена приложения может работать один или несколько потоков.

Домен приложения полностью изолирует используемые в его рамках ресурсы: как от других доменов того же процесса, так и от доменов других процессов. Домены приложения не могут совместно использовать никакие данные (будь то глобальные переменные или статические поля), за исключением случаев, когда используется протокол удаленного доступа к данным .NET. Надо сказать, что домены приложений .NET по своей роли очень похожи на апартаменты (apartment) в COM, Место доменов приложений в общей архитектуре .NET представлено на рис. 6.38,

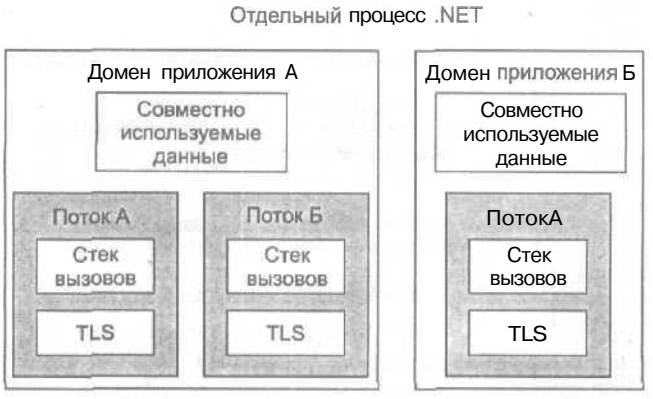


Рис. 6.38. Процесс .NET может содержать один или несколько доменов приложений. Домен приложений может содержать один или несколько потоков

Домены приложений программно реализуются при помощи типа System.AppDomain. Наиболее важные члены этого типа представлены в табл. 6.2.

Таблица 6.2. Наиболее важные члены AppDomain

Член	Назначение
CreateDomain()	Статический метод для создания нового домена приложения в текущем процессе
GetCurrentThreadId()	Этот статический метод возвращает идентификатор текущего потока
Unload()	Еще один статический метод — для выгрузки указанного домена приложения
BaseDirectory	Свойство, возвращающее базовый каталог, используемый распознавателем для поиска нужных приложению сборок
CreateInstance()	Создает экземпляр указанного типа, определенного в указанном файле сборки
ExecuteAssembly()	Запускает на выполнение сборку, имя которой указано в качестве параметра
GetAssemblies()	Возвращает список сборок, загруженных в текущий домен приложения
Load()	Загружает сборку в домен приложения

Работаем с доменами приложений

Как мы уже говорили, домен приложения — это дополнительный уровень в .NET между процессом и потоками. Чтобы показать применение доменов приложения на практике, рассмотрим следующий пример:

```
namespace MyAppDomain
{
    using System;
    using System.Windows.Forms;

    // Это пространство имен требуется для работы с типом Assembly
    using System.Reflection;

    public class MyAppDomain
    {
        public static void PrintAllAssemblies()
        {
            // Получаем список всех загруженных в текущий домен приложения
            // сборок
            AppDomain ad = AppDomain.CurrentDomain;
            Assembly[] loadedAssemblies = ad.GetAssemblies();
            Console.WriteLine("Here are the assemblies loaded in " + "this
                                                        appdomain\n");

            // Теперь выводим полное имя для каждой сборки
            foreach(Assembly a in loadedAssemblies)
            {
                Console.WriteLine(a.FullName);
            }
        }
        public static int Main(string[] args)
        {
            // Производим принудительную загрузку System.Windows.Forms.dll
            MessageBox.Show("Loaded System.Windows.Forms.dll");
```

```
            PrintAllAssemblies();
            return 0;
        }
    }
}
```

Прежде всего обратите внимание, что мы впервые используем новое пространство имен — System.Reflection. Это пространство имен будет подробно рассматриваться в следующей главе, а сейчас нам важно отметить, что в нем определен тип Assembly, который нам нужен для использования в методе PrintAllAssemblies().

Наш статический метод PrintAllAssemblies() получает ссылку на текущий домен приложения и выводит на консоль список всех загруженных сборок. Чтобы было интереснее, одну из сборок (System.Windows.Forms.dll) мы загрузили, намеренно использовав в Main() один из определенных в ней типов — MessageBox. Результат работы нашей программы представлен на рис. 6.39.

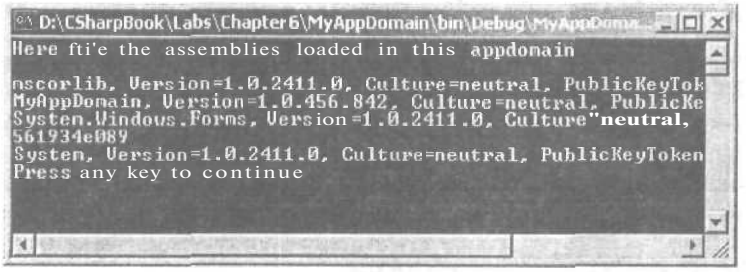


Рис. 6.39. Выводим список загруженных в домен приложения сборок

Код приложения MyAppDomain можно найти в подкаталоге Chapter 6.

Пространство имен System.Threading

Пространство имен System.Threading определяет большое число типов, которые используются для создания многопоточных приложений. Помимо типов для работы с конкретным потоком, в этом пространстве имен имеются типы для работы с набором потоков (ThreadPool), синхронизированным доступом к общим данным, а также простой (без графического интерфейса) класс Timer. Некоторые наиболее важные типы этого пространства имен представлены в табл. 6.3.

Таблица 6.3. Некоторые типы пространства имен System.Threading

Тип	Назначение
Interlocked	Для синхронизированного доступа к общим данным
Monitor	Обеспечивает синхронизацию потоковых объектов при помощи блокировок и управления ожиданием
Mutex	Примитив синхронизации, используемый для синхронизации разных процессов
Thread	Представляет поток, работающий *в* среде выполнения .NET. При помощи этого типа можно порождать в текущем домене приложения новые потоки

продолжение

Таблица 6.3 *(продолжение)*

Тип	Назначение
ThreadPool	Используется для управления набором взаимосвязанных потоков
Timer	Определяет делегат, который будет вызван в указанное время. Операция ожидания выполняется потоком в пуле потоков
WaitHandle	Представляет во время выполнения все объекты синхронизации (которые позволяют многократное ожидание)
ThreadStart	Представляет делегат со ссылкой на метод, который должен быть выполнен перед запуском потока
TimerCallback	Делегат для объектов Timer
WaitCallback	Делегат, который представляет метод обратного вызова для рабочих элементов ThreadPool

Работа с классом Thread

Самый простой тип в пространстве имен System.Threading — это класс Thread. Этот класс в .NET — не более чем объектная оболочка вокруг некоторого этапа выполнения программы внутри домена приложения. В этом типе предусмотрено значительное число членов (как статических, так и обычных) для создания текущим потоком новых потоков, а также для приостановки, полной остановки и удаления указанного потока. Наиболее важные статические члены этого типа представлены в табл. 6.4.

Таблица 6,4. Статические члены типа Thread

Статический член	Назначение
CurrentThread	Это свойство только для чтения возвращает ссылку на поток, выполняемый в настоящее время
GetData() SetData()	Возвращает/устанавливает значение для указанного слота в текущем потоке
GetDomain() GetDomainID()	Возвращает ссылку на домен приложения (идентификатор домена приложения), в рамках которого работает указанный поток
Sleep()	Приостанавливает выполнение текущего потока на указанное пользователем время

Обычные члены (обращение к которым производится через объект класса Thread) представлены в табл. 6.5.

Таблица 6.5. Обычные члены типа Thread

Член	Назначение
IsAlive	Это свойство возвращает «true» или «false» в зависимости от того, запущен поток или нет
IsBackground	Свойство предназначено для получения или установки значения, которое показывает, является ли этот поток фоновым
Name	Свойство для установки дружественного текстового имени потока
Priority	Позволяет получить/установить приоритет потока (используются значения из перечисления ThreadPrority)

Член	Назначение
ThreadState	Возвращает информацию о состоянии потока (используются значения из перечисления ThreadState)
Interrupt()	Прерывает работу текущего потока
Join()	Ждет появления другого потока (или указанный промежуток времени) и завершается
Resume()	Продолжает работу после приостановки работы потока
Start()	Начинает выполнение потока, определенного делегатом ThreadStart
Suspend()	Приостанавливает выполнение потока. Если выполнение потока уже приостановлено, то игнорируется

Запуск вторичных потоков

Если мы решили использовать в нашем приложении дополнительные потоки, нам потребуется класс Thread и специальный делегат для работы с потоками, называемый ThreadStart. Сам процесс создания дополнительных потоков в приложении очень прост. Прежде всего нам необходимо решить, какая функция будет выполнять у нас фоновую работу. Для большей наглядности мы создадим в нашем примере специальный вспомогательный класс, выводящий на консоль набор символов при помощи метода DoSomeWork():

```
internal class WorkerClass
{
        public void DoSomeWork()
        {
                // Выводим на консоль информацию о рабочем потоке
                Console.WriteLine("ID of worker thread is: {0}",
                                                Thread.CurrentThread.GetHashCode());

                // Выполняем некоторые действия
                Console.Write("Worker says: ");
                for(int 1 = o: i < 10; i++)
                {
                        Console.Write(i+ ", ");
                }
                Console.WriteLine();
        }
}
```

Теперь предположим, что у нас есть еще один класс — MainClass, в котором создается новый объект класса WorkerClass. При этом пусть согласно техническому заданию нам необходимо сделать так, чтобы все свои действия WorkerClass выполнял в рамках отдельного потока. Это можно сделать следующим образом (обратите внимание на применение типов Thread и ThreadStart):

```
publicclass MainClass
{
        public static int Main(string[] args)
        {
                // Выводим на консоль информацию о текущем потоке
                Console.WriteLine("ID of primary thread is: {0}",
                                                Thread.CurrentThread.GetHashCode());

                // Создаем объект класса WorkerClass
```

```
            WorkerClass w = new WorkerClass();

            // А теперь создаем и запускаем фоновый поток
            Thread backgroungThread = new Thread(new ThreadStart(w.DoSomeWork));

            backgroundThread.Start();

            return 0;
        }
}
```

Результат работы нашего приложения представлен на рис. 6.40. Обратите внимание: у нас действительно работают два отдельных потока, обладающие уникальными идентификаторами.

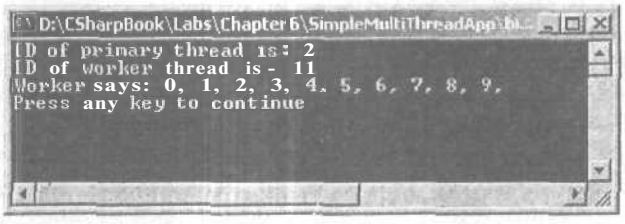


Рис. 6.40. Создание дополнительного потока

Именованные потоки

Одна из интересных особенностей класса Thread заключается в том, что этот класс позволяет присваивать потоку дружественное текстовое имя. Для этого используется свойство Name. Например, наш класс MainClass может выглядеть следующим образом:

```
public class MainClass
{
        public static int Main(string[] args)
        {
            // Присваиваем имя текущему потоку
            Thread primaryThread = Thread.CurrentThread;
            primaryThread.Name = "Boss man";

            Console.WriteLine("Id of {0} is {1}", primaryThread.Name,
                                        primaryThread.GethashCode());
            // Далее тот же код, что и раньше
            ...
        }
}
```

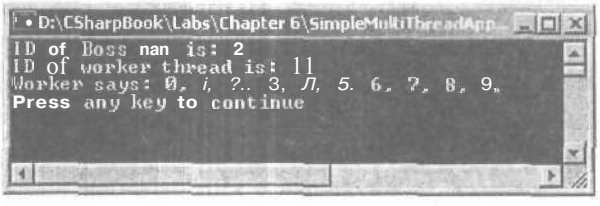


Рис. 6.41. Именованный поток

Результат работы этого варианта программы представлен на рис. 6.41.

Конечно же, это свойство используется главным образом для того, чтобы можно было легче ориентироваться в потоках нашей системы.

Параллельная работа потоков

В предыдущем примере мы создали в нашем приложении дополнительный поток. Однако убедиться в том, что использование двух потоков действительно создает иллюзию многозадачности (или обеспечивает реальную многозадачность — на многопроцессорных системах), нам не удалось по простой причине — вывод 10 цифр на консоль занимает слишком мало времени. За это время нам не удастся убедиться в том, что первичный поток приложения также продолжает работу. Чтобы доказать этот факт наглядно, мы изменим определение класса WorkerClass таким образом, чтобы он выводил на конеоль не 10 цифр, а 30 000 и для каждой из них использовал метод WriteLine() (а не Write()):

```
internal class WorkerClass
{
        public void DoSomeWork()
        {
                ...
                // Выполняем большую работу
                Console.Write("Worker says: ");
                fordnt 1-0: i < 30000; i++)
                {
                        Console.WriteLine(i + ~. ");
                }
                Console.WriteLine();
        }
;
```

Теперь мы изменим MainClass таким образом, чтобы сразу после создания второго потока выводилось окно сообщения:

```
public class MainClass
{
        public static int Main(string[] args)
        {
                // Присваиваем имя текущему потоку
                ...
                // Создаем объект класса WorkerClass
                ...
                // Создаем дополнительный поток
                ...

                // Пока работает фоновый поток, наша программа будет заниматься другими
                // делами
                MessageBox.Show("I 'm busy");

            -  return 0;
        }
}
```

При запуске приложения мы увидим, что во время работы фонового потока (который выводит длинный столбик цифр) появится окно сообщения, которое можно, к примеру, перетаскивать по экрану мышью (рис. 6.42).

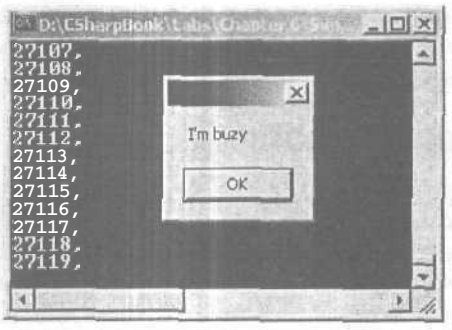


Рис. 6.42. Работают два активных потока

Однопоточное приложение будет вести себя совершенно по-другому. Давайте изменим метод Main() таким образом, чтобы при запуске приложения мы смогли определить, какое количество потоков будет создано:

```
public static int Main(string[] args)
{
        Console.Write("Do you want [1] or [2] threads? ");
        string threadCount = Console.ReadLine();

        // Присваиваем имя текущему потоку
        ...

        // Создаем объект класса WorkerClass
        WorkerClasss w = new WorkerClass();

        // Создаем дополнительный поток, только если это указано пользователем
        if(threadCount - = "2")
        {
                // Создаем дополнительный поток
                Thread backgrounjThread = new Thread(new ThreadStart(w.DoSomeWork));
                backgroundThread.Start();
        }
        else
                w.DoSomeWork();

        // Даем первичному потоку свое задание
        MessageBox.Show("I'm busy!");

        return 0;
}
```

Если пользователь выбрал 1 (что означает «один поток»), то ему придется подождать, пока не будут показаны все 30 000 чисел. Только после этого будет показано окно сообщения. Если же пользователь ввел 2 и выбрал, таким образом, многопоточность, он сможет работать с окном сообщения и в то время, когда на консоль будут выводиться числа.

Как «усыпить» поток

Для того чтобы приостановить выполнение потока на определенное время, можно использовать статический метод Thread.Sleep(). Время, на которое поток должен

«заснуть», указывается при этом в миллисекундах. Для примера, давайте снова внесем изменения в класс WorkerClass. Теперь метод DoSomeWork() будет выводить уже не 30 000 чисел, а всего 5, Однако между выводом каждого числа поток, который этим занимается, будет «погружаться в сон» на 5 секунд.

```
internal class WorkerClass
{
        public void DoSomeWorkO
        {
                // Выводим информацию о рабочем потоке
                Console.WriteLine("ID of worker thread is: {0}",
                                Thread.CurrentThread.GetHashCode());

                // Делаем работу "с перекурами"
                Console.Write("Worker says: ");
                for(int i = 0; 1 < 5; i++)
                {
                        Console.WriteLine(i + ", ");
                        Thread.Sleep(5000);
                }
                Console.WriteLine();
        }
}
```

Результат работы программы представлен на рис. 6 43.

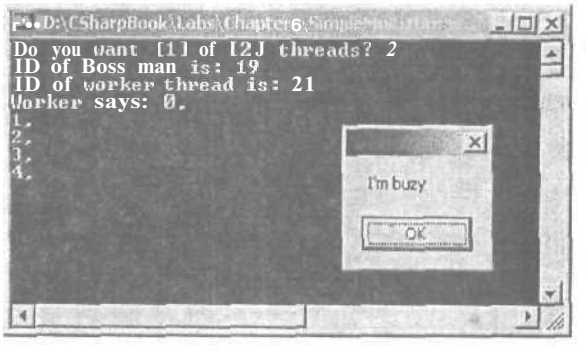


Рис. 6.43. Два активных процесса, один из которых время от времени «засыпает»

Код приложения SimpleMultiThreadApp можно найти в подкаталоге Chapter 6.

Одновременный доступ к данным из разных потоков

Познакомившись с предыдущими примерами, вы можете решить, что потоки — это чудодейственное средство, которое может почти все. Однако помните, что главная цель использования нескольких потоков — это повышение производительности приложения и удобства работы пользователей. При этом потоки следует использовать осторожно. Если в нашем приложении будет слишком много потоков, то производительность его работы может даже снизиться — за счет того, что процессору придется тратить много времени на переключение между потоками. Однако гораздо более серьезные проблемы могут возникнуть при одновременном доступе нескольких потоков к одним и тем же общим данным.

В нашем примере из предыдущих разделов никакой порчи данных произойти не может. Однако представьте себе, что и первичный, и дополнительный потоки изменяют один и тот же блок данных. Планировщик операционной системы выделяет потокам время центрального процессора в соответствии со своим внутренним алгоритмом, ничего не зная о целостности данных. В результате может получиться так, что первый поток начнет изменять данные и будет приостановлен, а в это время за дело возьмется другой *поток*. Очень вероятно, что результат будет печален — данные будут безнадежно испорчены.

Это несложно проиллюстрировать на примере. Правда, для простоты в нашем случае мы при помощи двух потоков будем портить вывод данных не в файл, а на системную консоль, но принципиально это ничего не меняет. Мы создадим новое многопоточное консольное приложение C# с именем MultiThreadSharedData. В этом приложении будет использован класс WorkerClass, практически такой же, как и в предыдущем примере:

```
Internal class WorkerClass
t
        public void DoSomeWork()
        {
                // Выполняем привычную работу
                for(int 1 - 0: i < 5: i++)
                {
                        Console.WriteLine("Worker says: {0}.", i):
                }
        }
}
```

В приложении такжебудет еще один класс — MainClass. В этом варианте MainClass будет создавать три отдельных дополнительных потока. Каждый из этих потоков будет совместно использовать один и тот же объект класса WorkerClass:

```
publicclass MainClass
{
        public static int Main(string[] args)
        {
                // Создаем единственный объект класса WorkerClass
                WorkerClass w = new WorkerClass():

                // Создаем три отдельных потока, каждый из которых производит вызов
                // к одноиу и тону же объекту
                Thread workerThreadA = new Thread(new ThreadStart(w.DoSomeWork));
                Thread workerThreadB = new Thread(new ThreadStart(w.DoSomeWork));
                Thread workerThreadC = new Thread(new ThreadStart(w.DoSomeWork));

                // Теперь запускаем все три потока
                WorkerThreadA.Start();
                WorkerThreadB. Start();
                WorkerThreadC.Start();

                return 0:
        }
}
```

Перед тем как запустить это приложение, давайте разберемся в существе проблемы. Первичный поток домена приложения запускает три дополнительных потока. Все три потока совместно используют одни и те же данные — единственный объект класса WorkerClass. Поскольку мы не используем никакой блокировки ресурсов, скорее всего, вывод на системную консоль будет испорчен: один процесс

начинает вывод на консоль, в его вывод вклинивается другой, их обоих оттесняет третий и т. д. Результат непредсказуем. Три разных варианта того, что может получиться в результате работы этого приложения, представлены на рис. 6.44-6.46.

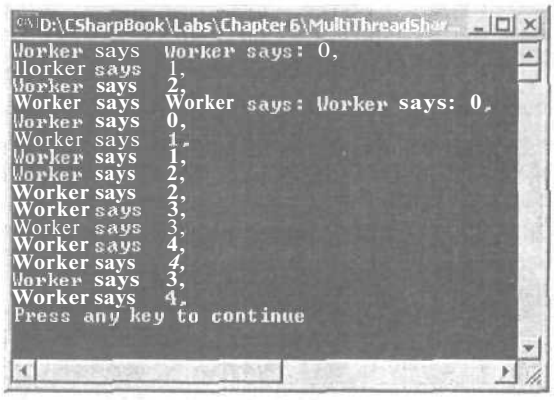


Рис. 6.44. Вывод данных испорчен: потоки мешают друг другу

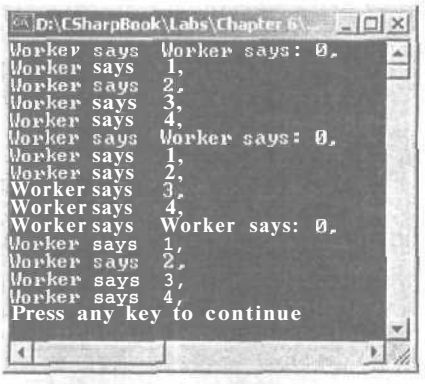


Рис. 6.45. Пожалуй, так еще хуже. Соперничество потоков приводит к непредсказуемым результатам

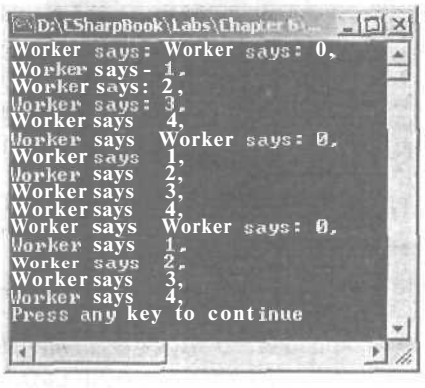


Рис. 6.46. Возможен и такой вариант. Вывод все равно испорчен

Для решения подобных проблем в C# предусмотрены средства программной синхронизации доступа к общим данных. Их мы и рассмотрим в следующих разделах.

Ключевое слово lock

Первый способ синхронизации доступа к методу DoSomeWork() объекта WorkerClass — воспользоваться ключевым словом lock. Это ключевое слово позволяет заблокировать блок кода таким образом, что в каждый отдельный момент времени его сможет использовать только один поток. Всем остальным потокам придется ждать, пока поток, занявший этот блок кода, закончит свою работу. Применение ключевого слова lock производится очень просто:

```
internal class WorkerClass
{
        public void DoSomeWork()
        {
                // Только один поток в конкретный ноиент времени сможет выполнять этот
                // код!
                lock(this)
                {
                        // Делаем все ту же работу
                        forCint i=0; i < 5: i++)
                        t
                                Console.WriteLine("Worker says: {0}.", i);
                        }
                }
        }
}
```

Если мы запустим наш пример с измененным классом WorkerClass, мы увидим, что теперь вывод на консоль производится исключительно аккуратно (рис. 6.47).

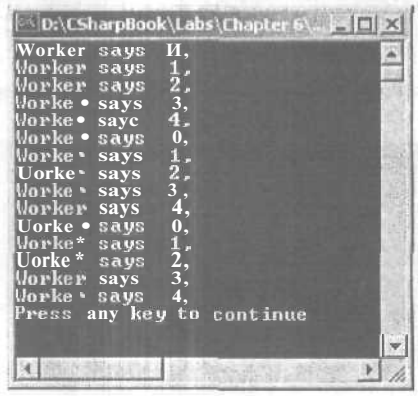


Рис. 6.47. Между потоками царит гармония

Можно считать, что ключевое слово lock в C# — аналог примитива CRITICAL_SECTION и соответствующих вызовов API в традиционных приложениях Win32.

Код приложения MultiThreadSharedData можно найти в подкаталоге Chapter 6.

Использование System.Threading.Monitor

Ключевое слово lock в C# — это всего лишь сокращение для синтаксической конструкции с применением класса System.Threading.Monitor. Такая синтаксическая конструкция, заменяющая lock, в нашем случае могла бы выглядеть следующим образом:

```
internal class WorkerClass
{
    public void DoSomeWork()
    {
        // Определяем элемент для мониторинга в целях синхронизации
        Monitor.Enter(this);
        try
        {
            // Выполнить работу...
            for(int i = 0; i < 5; i++)
            {
                Console.WriteLine("Worker says: {0}.", i);
            }
        }
        finally
        {
            // Была ошибка или нет, а из монитора придется выйти
            Monitor.Exit(this);
        }
    }
}
```

Если мы запустим программу с этим вариантом класса WorkerClass, то результат не изменится по сравнению с рис. 6.47 (что не так и плохо). Обратите внимание на применение статических методов Enter() и Exit() для входа и выхода из участка кода с блокировкой.

Применение System.Threading.Interlocked

В пространстве имен System.Threading предусмотрен также специальный тип, который позволяет увеличивать или уменьшать значение переменной на единицу потокобезопасным способом. Чтобы объяснить суть проблемы, проиллюстрируем ее на примере класса, который будет называться IHaveNoIdea. Пусть в этом классе будет внутренний счетчик ссылок refCount. Один из методов данного класса будет ответственен за приращение этого счетчика на единицу, а другой — за уменьшение его значения на ту же единицу (звучит знакомо, не правда ли?):

```
public class IHaveNoIdea
{
    private long refCount = 0;

    public void AddRef()
    { ++refCount; }

    public void Release()
    {
        if(--refCount == 0)
        {
            GC.Collect();
        }
    }
}
```

Предположим, что в текущем домене приложения существует несколько потоков, которые делают вызовы AddRef() и Release(). При этом расписание работы потоков вполне может сложиться таким образом, что значение refCount может стать отрицательным. Результат с точки зрения реакции программы на такое неожиданное значение может быть самым удивительным.

Чтобы не допустить возникновения подобной ситуации, можно воспользоваться классом System.Threading.Interlocked. Обратите внимание на передачу методам Increment() и Decrement() ссылки на переменную refCount с применением ключевого слова ref:

```
public class IHaveNoIdea
{
        private long refCount - 0;

        public void AddRef()
        {
                Interlocked.Increment(ref refCount);
        }

        public void Release()
        {
                if(Interlocked.Decrement(ref refCount) = = 0)
                {
                        GC.Collect();
                }
        }
}
```

К этому моменту вы уже обладаете достаточной информацией, чтобы представлять опасность в мире многопоточных сборок. Мы советуем вам самостоятельно рассмотреть также остальные типы пространства имен System.Threading — вполне *возможно*, что они вам пригодятся,

Подведение итогов

В этой главе рассматривались особенности строения сборок .NET — файлов EXE и DLL, которые так не похожи на обычные EXE и DLL приложений Win32. Мы познакомились с метаданными типов, манифестом и кодом промежуточного языка. Кроме того, мы выяснили, в чем заключаются различия между частными сборками и сборками для общего пользования, а также разобрались с механизмом поиска сборок средой выполнения .NET и применением файлов конфигурации приложений.

Кроме того, в приложениях .NET появился новый уровень, который находится между процессами и потоками — уровень доменов приложений. Домены приложений могут включать в себя любое количество потоков. Используя типы пространства имен System.Threading, мы можем создавать многопоточные приложения, а также реализовывать потокобезопасный доступ к общим данным в многопоточных приложениях.

Рефлексия типов и программирование с использованием атрибутов

7

Как уже говорилось в предыдущей главе, приложения .NET состоят из сборок. Информацию о типах в сборках можно получить при помощи разных средств. Например, для этого можно использовать интегрированный Object Browser (Просмотрщик объектов) в среде разработки Visual Studio.NET. Еще одно знакомое нам средство — это утилита ILDasm.exe. Однако доступ к информации о типах сборки можно получить и *программными способами*. Для этого предназначены типы, определенные в пространстве имен `System.Reflection`, и именно об этом пойдет речь в этой главе.

После того как мы познакомимся с исследованием сборок в процессе выполнения — рефлексией типов, мы перейдем к рассмотрению тесно связанных с этой темой возможностей. Типы, определенные в пространстве имен `System.Reflection.Emit`, позволяют создавать сборки прямо в процессе выполнения программы — «на лету». Мы также убедимся, что применение позднего связывания может оказаться очень полезным и в приложениях .NET (как мы увидим в следующих главах, позднее связывание очень важно для организации взаимодействия приложений .NET и традиционных приложений COM).

Глава заканчивается описанием того, как можно использовать внутри сборок .NET пользовательские метаданные путем применения встроенных и специально созданных нами атрибутов. Если вы пришли из мира COM, то вы увидите, что многие из полезных черт IDL нашли свое применение и дальнейшее развитие в .NET.

Что такое рефлексия типов

В мире .NET рефлексия — это процесс обнаружения типов во время работы программы. Используя рефлексию, мы можем во время выполнения загрузить сборку и получить о ней ту же информацию, которую показывает утилита ILDasm.exe. Например, мы можем получить список всех типов, определенных в указанном нами

модуле, а также всех членов каждого из типов — всех методов, полей, свойств и событий. Мы можем прямо в процессе работы программы получить перечень всех интерфейсов, поддерживаемых классом или структурой, параметры методов и прочую информацию, важную для объектно-ориентированного программирования.

Главные элементы, которые необходимы для использования возможностей рефлексии в наших программах — это класс Type из пространства имен System и типы пространства имен System.Reflecti on. Класс System. Type содержит большое количество методов, которые можно использовать для получения ценной информации о самых разных типах. Пространство имен System.Reflection определяет типы для организации позднего связывания и динамической загрузки сборок. Вначале речь пойдет о System.Type.

Класс Type

Большая часть возможностей, которые дает пространство имен System.Reflection, связана с применением абстрактного типа System. Type. Этот класс содержит значительное количество методов, которые могут быть использованы для получения информации о типах в нашей программе. Полный список всех членов занял бы очень много места, поэтому в табл. 7.1 мы приводим перечень лишь наиболее важных свойств и методов этого класса.

Таблица 7.1. Члены класса Type

Член	Назначение
IsAbstract IsArray IsClass IsCOMObject IsEnum IsInterface IsPrimitive IsNestedPublic IsNestedPrivate IsSealed IsValueType	Эти свойства (как и многие другие) позволяют определить основные характеристики конкретного типа в программе (например, является ли он абстрактным, является ли он массивом, является ли он классом и т. п.)
GetConstructors() GetEvents() GetFields() GetInterfaces() GetMethods() GetMembers() GetNestedTypes() GetProperties()	Эти методы (и аналогичные им) возвращают массив с набором интересующих пользователя элементов (конструкторами, событиями, полями и т. п.). Каждый метод возвращает массив соответствующих типов, например, GetFields() возвращает массив типов FieldInfo; GetMethods() — массив типов MethodInfo и т. д. Для каждого из этих методов есть парный ему (например, GetMethod() вместо GetMethods(), GetProperty() вместо GetProperties()) для работы только с одним элементом вместо массива со всеми элементами. Например, передав методу GetMethod() имя конкретного метода, можно получить о нем информацию
FindMembers()	Возвращает массив типов MemberInfo основываясь на заданных критериях поиска
GetType()	Этот метод возвращает объект типа Type по строковому имени
InvokeMember()	Этот метод используется для позднего связывания указанного элемента

Получение объекта класса Type

Получить объект класса Type можно самыми разными способами. Однако при этом воспользоваться стандартным подходом — создать объект класса Type напрямую

при помощи ключевого слова new мы не сможем. Причина проста: System.Type - это абстрактный класс.

Первый способ получения объекта класса Type — применить метод GetType(), определенный в классе System.Object (как мы помним, все остальные типы C# являются производными от этого класса):

```
// Получаем объект класса Type из имеющегося обьекта класса Foo
Foo theFoo = new Foo();
Type t = theFoo.GetType();
```

Второй способ — использовать статический метод GetType(), определенный в самом классе System.Type. Этому методу нужно будет передать текстовое имя того типа, который нас интересует:

```
// Получаем объект класса Type, используя статический метод GetTypeO
// из самого класса Type
Type t = null;
t = Type.GetType("Foo");
```

Третий способ — использовать ключевое слово typeof:

```
// Получаем объект класса Type при помощи кллючевого слова typeof
Type t = typeof(Foo);
```

Обратите внимание, что при использовании Type.GetType() и typeof нет необходимости создавать объект типа для того, чтобы получить информацию об этом типе,

А теперь, после того как мы создали объект класса Type, посмотрим, что это нам дает,

Возможности класса Type

Проиллюстрируем возможности System.Type на примере класса Foo, определение которого приведено ниже (реализации методов в этом примере могут быть любыми):

```
// Мы сможем получить разнообразную информацию об этом классе во время выполнения
namespace TheType
{
// Два интерфейса
public interface IFaceOne
{ void MethodA(); }

public interface IFaceTwo
{ void MethodB(); }

// Класс Foo поддерживает эти два интерфейса
public class Foo: IFaceOne, IFaceTwo
{
        // Поля
        public int myIntField;
        public string myStringField;

        // Метод
        public void myMethod(int p1, string p2) {...}

        // Свойство
        public int MyProp
```

```
        {
            get { return myIntField; }
            set { myIntField = value; }
        }

        // Методы интерфейсов IFaceOne и IFaceTwo
        public void MethodA() {...}
        public void MethodB() {...}
    }
}
```

А теперь мы определим класс FooReader — программу, которая сможет получать во время выполнения информацию о классе Foo — его методах, свойствах, поддерживаемых интерфейсах и полях (и кое-что еще). В классе FooReader определено несколько статических методов, которые очень похожи друг на друга. Первым идет метод ListMethods(), который позволяет получить информацию о всех методах класса Foo при помощи объекта System.Type. Обратите внимание, что Type.ListMethods() возвращает массив типов MethodInfo:

```
// Получаем информацию об именах всех методов Foo
public static void ListMethods(Foo f)
{
    Console.WriteLine("***** Methods of Foo *****");

    Type t = f.GetType();
    MethodInfo[] mi = t.GetMethods();
    foreach(MethodInfo m in mi)
        Console.WriteLine("Method:{0}", m.Name);

    Console.WriteLine("**************************\n");
}
```

Метод ListFields() очень похож на ListMethods. Отличие заключается в том, что мы получаем список полей, а не методов и работаем с массивом типов FiledInfo вместо MethodInfo:

```
// Получаем информацию об именах всех полей Foo
public static void ListFields(Foo f)
{
    Console.WriteLine("***** Fields of Foo *****");

    Type t = f.GetTypeO;
    FieldInfo[] fi = t.GetFields();
    foreach(FieldInfo field in fi)
        Console.WriteLine("Field:{0}", field.Name);

    Console.WriteLine("**************************\n");
}
```

Методы ListVariosStats(), ListProps() и ListInterfaces(), видимо, объяснять уже не надо:

```
// Выводим разную информацию о Foo
public static void ListVariosStats(Foo f)
{
    Console.WriteLine("***** Various stats about Foo *****");
    Type t = f.GetTypeO:

    Console.WriteLine("Full name is: {0}", t.FullName);
```

```
        Console.WriteLine("Base is: {0}", t.BaseType);
        Console.WriteLine("Is it abstract? {0}", t.IsAbstract);
        Console.WriteLine("Is it a COM object? {0}", t.IsCOMObject);
        Console.WriteLine("Is it sealed? {0}", t.IsSealed);
        Console.WriteLine("Is it a class? {0}", t.IsClass);

        Console.WriteLine("*************************\n");
    }

    // Выводим список всех свойств
    public static void ListProps(Foo f)
    {
        Console.WriteLine("***** Properties of Foo *****");

        Type t = f.GetType();
        PropertyInfo[] pi = t.GetProperties();
        foreach(PropertyInfo prop in pi)
            Console.WriteLine("Prop: {0}", prop.Name);

        Console.WriteLine("*************************\n");
    }

    // Выводим список всех интерфейсов, поддерживаемых Foo
    public static void ListInterfaces(Foo f)
    {
        Console.WriteLine("***** Interfaces of Foo *****");

        Type t = f.GetTypeO;
        Type[] ifaces = t.GetInterfaces();
        foreach(Type i in ifaces)
            Console.WriteLine("Interface: {0}", i.Name);

        Console.WriteLine("*************************\n");
    }
```

Метод Main() в классе FooReader должен просто вызвать каждый из этих статических методов:

```
// Кладем Foo под "увеличительное стекло"
namespace TheType
{
using System;

// Требуется для использования MethodInfo, FieldInfo и т.д.
using System.Reflection;

public class FooReader
{
    // Здесь - определения статических методов, приведенные выше
    ...

    public static int Main(string[] args)
    {
        // Создаем новый объект класса Foo
        Foo theFoo = new Foo();

        // А теперь получаем всю информацию о классе Foo
        ListVariouStats(theFoo);
```

```
            ListMethods(theFoo);
            ListFields(theFoo);
            ListProps(theFoo);
            ListInterfaces(theFoo);
            return 0;
        }
}
}
```

Результат работы программы представлен на рис. 7.1.

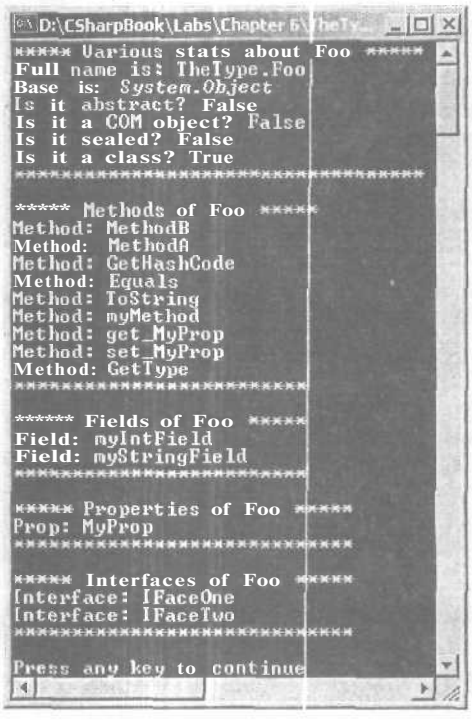


Рис. 7.1. Рефлексия для класса Foo

Интересно, не правда ли? Класс System.Type позволяет получать очень ценную информацию о типах в наших программах. Однако, как мы увидим, типы пространства имен System.Reflection позволяют еще больше расширить возможности рефлексии в C#.

Код приложения TheType можно найти в подкаталоге Chapter 7.

Типы пространства имен System.Reflection

Как и любое другое пространство имен, System.Reflection содержит набор типов, обладающих схожей областью применения. Наиболее важные из этих типов представлены в табл. 7.2. С применением некоторых из них мы уже познакомились в нашем предыдущем примере.

Таблица 7.2. Некоторые типы пространства имен System.Reflection

Тип	Назначение
Assembly	Этот класс (вместе с многочисленными взаимосвязанными типами) содержит методы для загрузки и изучения сборки, а также выполнения с ней различных операций
AssemblyName	Этот класс позволяет получать информацию об идентификации сборки (информацию о версии, естественном языке и т. п.)
EventInfo	Хранит информацию о событии
FieldInfo	Хранит информацию о поле
MemberInfo	Абстрактный базовый класс, определяющий общие члены для EventInfo, FieldInfo, MethodInfo и PropertyInfo
MethodInfo	Хранит информацию о методе
Module	Позволяет обратиться к модулю в многофайловой сборке
ParameterInfo	Хранит информацию о параметре
PropertyInfo	Хранит информацию о свойстве

Загрузка сборки

Обычно при использовании типов из System.Reflection значительную часть работы выполняет класс Assembly. При помощи этого класса мы можем динамически загружать сборки, обращаться к членам класса в процессе выполнения (позднее связывание), а также получать информацию о самой сборке.

Первое, что нам потребуется сделать для получения информации о сборке, — загрузить эту сборку в память. Давайте создадим новый консольный проект C# с именем CarReflection, который будет производить загрузку сборки CarLibrary (она была создана нами в предыдущей главе). Для загрузки сборки используется статический метод Assembly.Load(), которому передается дружественное текстовое имя сборки:

```
// Получаем информацию о сборке CarLibrary
namespace CarReflector
{
using System;
using System.Reflection;
using System.IO;              // Нужно для использования FileNotFoundException

public class CarReflector
{
        public static int Main(string[] args)
        {
                // Используем метод Assembly.Load() для загрузки сборки
                Assembly a = null;
                try
                {
                        a=Assembly.Load("CarLibrary");
                }
                catch(FileNotFoundException e)
                {Console.WriteLine(e.Message);}

                return 0;
        }
}
}
```

Обратите внимание, что в нашем примере мы передаем методу Assembly.Load() только дружественное текстовое имя сборки, которую нужно загрузить в память. Однако, как вы, наверное, уже догадываетесь, существует множество перегруженных вариантов этого метода, которые принимают и другую информацию, которая может оказаться необходимой. Методу Assembly.LoadO можно передать в качестве параметра кроме текстового имени еще и номер версии, значение открытого ключа, информацию о естественном языке, использованном в сборке, и «сильное имя» сборки.

Весь вышеперечисленный набор элементов, характеризующих сборку, носит общее название «отображаемого имени» (display name). Формат этого имени выглядит как текстовое имя, за которым через запятую следуют необязательные остальные элементы (они могут быть приведены в любом порядке). Шаблон отображаемого имени выглядит следующим образом:

```
Имя (,Loc - Информация_о_естественном_языке) (,Ver = Основная.Дополнительная.Редакция.
Сборка) (,SN = Сильное имя)
```

SN = null означает, что загружаемая сборка — частная (а не для общего доступа). Loc="" (пустое значение для естественного языка) значит, что используется естественный язык по умолчанию. Например:

```
// Загрузка сборки с использованием отображаемого имени
a = Assembly.Load(@"CarLibrary, Ver=1.0.454.30104, SN=null, Loc="");
```

Кроме того, в System.Reflection предусмотрен специальный класс AssemblyName, который можно передавать Assembly.Load() в качестве параметра (вместо отображаемого имени). Выглядеть это может так:

```
// Такое "объектно-ориентированное" имя сборки
AssemblyName asmName;
asmName = new AssemblyName O;
asm.Name - "CarLibrary";

Version v - new Version("1.0.454.30104");
asmName.Version = v;

a = Assembly.Load(asmName);
```

Вывод информации о типах в сборке

Теперь, когда мы загрузили сборку CarLibrary, можно получить информацию об имени каждого типа, определенного в этой сборке. Для этого используется метод Assembly.GetTypes():

```
public class CarReflector
{
        public static int Main(string[] args)
        {
                Assembly a = null;
                try
                {
                        a=Assembly.Load("CarLibrary");
                }
                catch(FileNotFoundException e)
```

```
            {Console.WriteLine(e.Message);}

            ListAllTypes(a);
            return 0;
        }

        // Выводим информацию о всех типах в сборке
        private static void ListllTypes(Assembly a)
        {
            Console.WriteLine("Listing all types in {0}", a.FullName);
            Type[] types = a.GetTypes();
            foreach(Type t in types)
                Console.WriteLine("Type: {0}", t);
        }
}
```

Вывод информации о членах класса

Предположим, что нам потребовалось получить при выполнении программы информацию о всех членах одного из классов CarLibrary (пусть это будет класс Hi ni Van). Сделать это очень просто — достаточно воспользоваться методом GetMembers(), определенным в классе System. Type, как показано ниже. В этом классе предусмотрены также методы GetProperties(), GetMethods() и прочие, которые можно использовать для получения списка не всех членов класса, а только определенной их разновидности. Метод GetMembers() возвращает массив типов MemberInfo. Пример применения Type. GetMembers О может выглядеть так:

```
// Еще один статический метод класса CarReflector
private static void ListAllMembers(Assembly a)
{
    Type miniVan = a.GetType("CarLibrary.MiniVan");
    MemberInfo[] mi = miniVan.GetMembers();
    foreach(MemberInfo m in mi)
        Console.WriteLine("Type {0}: {1} ", m.MemberType.ToString(), m);
}
```

Результат работы программы представлен на рис. 7.2.

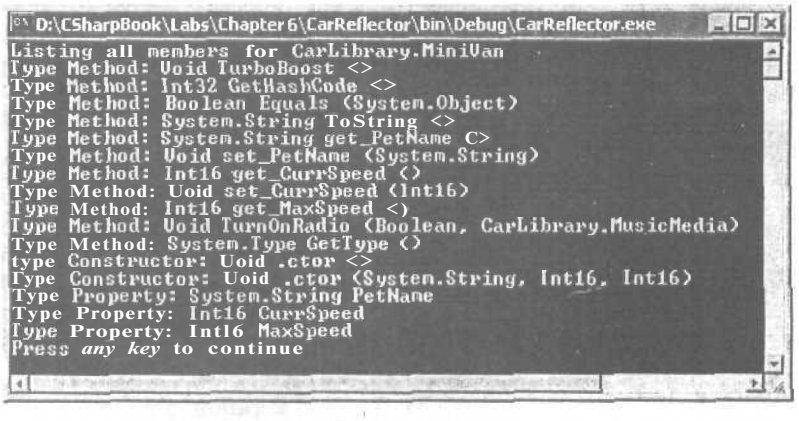


Рис. 7.2. Класс MiniVan «под микроскопом»

Вывод информации о параметрах метода

При помощи рефлексии **мы** можем спуститься зще на один уровень вниз и получить также информацию о параметрах конкретного метода. Например, предположим, что класс Car определяет еще один метод:

```
// Новый член класса Car
public void TurnOnRadio(bool state. MusicMedia mm)
{
        if(state)
                MessageBox.Show("Jamming  with  {0}", rni.ToStringO);
        else
                MessageBox.Show("Quiet time...");
}
```

Метод TurnOnRadio() принимает два параметра, при этом второй параметр — это пользовательское перечисление:

```
// Откуда будет звучать музыка
public enum MusIcMedIa
{
        musicCD,
        musicTape,
        musicRadio
}
```

Для извлечения информации о параметрах метода используется метод MethodInfo .GetParameters(). Этот метод возвращает массив объектов класса ParameterInfo. Каждый элемент в этом массиве содержит несколько свойств, которые позволяют полностью описать любые возможные характеристики параметра. Дополним класс CarReflector новым методом GetParams(), который иллюстрирует применение всех этих возможностей:

```
// Получаем информацию о параметрах для метода TurnOnRadio()
private static void GetParams(Assembly a)
{
        // Получаем тип MethodInfo
        Type miniVan - a.GetType("CarLibrary.MiniVan");
        MethodInfo mi = miniVan.GetMethod("TurnOnRadio");

        // Выводим информацию о количестве параметров
        Console.WriteLine("Here are the params for {0}", mi.Name);
        ParameterInfo[] myParams - mi.GetParameters();
        Console.WriteLine("Method has " + myParams._ength + " params");

        // Выводим некоторые характеристики каждого из параметров
        foreach(ParameterInfo pi in myParams)
        {
                Console.WriteLine("Param name: {0}", pi.Name);
                Console.WriteLine("Position in method: {0}", pi.Position);
                Console.WriteLine("Param type: {0}", pi.ParameterType);
        }
}
```

Результат работы этой программы представлен на рис. 7.3.

Код приложения CarReflector можно найти в подкаталоге Chapter 7.

Как мы видим, при помощи класса System.Type и типов, определенных в пространстве имен System.Reflection, непосредственно в процессе выполнения про-

граммы можно получить ту же информацию, что и при помощи ILDasm.exe. Главное отличие — в наших программах мы просто выводили всю полученную информацию на системную консоль, а в ILDasm.exe та же информация представлена при помощи иерархического интерфейса.

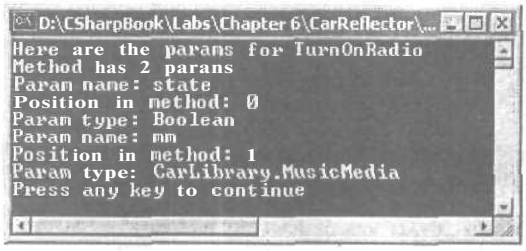


Рис. 7.3. Информация о параметрах метода

Применение позднего связывания

Пространство имен System.Reflection обеспечивает еще одну важную возможность — позднее связывание. Позднее связывание (late binding) — это технология, которая позволяет обнаруживать типы, определять их имена и члены непосредственно в процессе выполнения (в отличие от обычного раннего связывания, когда эти операции производятся во время компиляции). После того как наличие типа установлено, мы можем динамически вызывать его методы, получать доступ к свойствам, устанавливать значения полей и т. п.

Как правило, позднее связывание используется в достаточно сложных программах. Если у нас есть возможность обойтись ранним связыванием, то лучше так л сделать. Раннее связывание обеспечивает обнаружение возможных ошибок еще во время выполнения, и, как правило, его применение более надежно. Позднее связывание часто используется при создании средств разработки, а также для взаимодействия COM/.NET. Например, при помощи позднего связывания программист .NET может получить ссылку на интерфейс IDispatch для объекта COM. Подробно взаимодействие COM и .NET рассматривается далее в этой книге. А сейчас мы рассмотрим, как реализуется позднее связывание в C# на примере класса MiniVan.

Класс System.Activator

Главный тип, при помощи которого в .NET реализуется позднее связывание, — это класс System.Activator. В нем помимо методов, унаследованных от System.Object, содержится лишь несколько собственных членов. Самый важный из них — это метод Activator.CreateInstance(). Этот метод предназначен для создания объекта типа во время выполнения. Существует несколько перегруженных вариантов этого метода, обеспечивающих большую гибкость в использовании. Вариант CreateInstance(), который мы будем использовать, принимает готовый объект типа Type:

```
// Создаем объект выбранного нами типа "на лету"
public class LateBind
```

```
{
        public static int Main(string[] args)
        {
                // Используем класс Assembly для загрузки сборки
                Assembly a = null;
                try
                {
                        a = Assembly.Load("CarLibrary");
                }
                catch(FileNotFoundException e)
                {Console.WriteLine(e.Message);}

                // Получаем объект Type для класса MiniVan
                Type miniVan - a.GetTypeC'CarLibrary MiniVan");

                // Создаем объект класса MiniVan "на лету"
                object obj - Activator.CreateInstance(miniVan);
        }
}
```

В настоящий момент переменная obj указывает на объект класса MiniVan, который был создан при помощи метода Activator.CreateInstance. Теперь предположим, что нам надо вызвать метод TurboBoost() для этого объекта. Как мы помним, для автомобиля этого типа такое ускорение приводит к безвременной кончине двигателя (состояние egnState принимает значение dead) и выводу окна сообщения.

Поскольку мы используем позднее связывание, вызов метода для объекта — не такое простое дело. Вначале мы должны получить объект класса MethodInfo для метода TurboBoostO при помощи Type.GetMethod(). Затем мы можем использовать этот объект Methodlnfo, чтобы при помощи метода InvokeC) вызвать метод TurboBoost(). MethoInfo.InvokeC) требует, чтобы все параметры, которые нужно будет передать вызываемому методу, передавались Invoke() как массив объектов класса System.Object. Поскольку нашему TurboBoostO никакие параметры не нужны, мы можем просто передать значение типа null (оно как раз и значит, что вызываемому методу параметры не передаются);

```
public static int Main(string[] args)
{
        // Загружаем CarLibrary при помощи класса Assembly
        ...

        // Получаем объект типа Type
        Type miniVan = a.GetType("CarLibrary.MiniVan");

        // Создаем объект класса MiniVan "на лету"
        object obj - Activator.CreateInstance(miniVan);

        // Получаем объект класса MethodInfo для негода TurboBoostO
        MethodInfo mi - miniVan.GetMethod("TurboBoost");

        // Вызываем метод (передаем null вместо параметров)
        mi.invoke(obj, null);
        return 0;
}
```

Если все сделано правильно, мы увидим окно сообщения, подобное представленному на рис. 7.4.

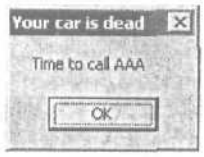


Рис. 7.4. Позднее связывание

А теперь предположим, что нам нужно при помощи позднего связывания вызвать следующий новый метод класса MiniVan;

```
// Проводим миротворческую операцию...
public void TellChildToBeQuiet(string kidName, int shameIntensity)
{
        for(int i - 0; i < numb; i++)
              MessageBox.Show("Be queit {0}!!, kidName);
}
```

Метод TellChi ldToBeQuet() принимает два параметра. В этом случае для вызова этого при помощи MethodInfo.Invoke() нам придется создать массив параметров:

```
// Так вызывается при помощи общего связывания метод с параметрами
object[] paramArray = new object[23:
paramArray[0] = "Fred";                  // Имя ребенка
paramArray[1] = 4;                       // Насколько сильно его надо устыдить
mi = miniVan.GetMethod("TellChi ldToBeQueit");
mi.Invoke(obj, paramArray);
```

При запуске этой программы сразу четыре окна сообщения будут призывать Фредерика к порядку (рис. 7.5).

Рис. 7.5. Вызов метода с параметрами при позднем связывании

Код приложения LateBinding можно найти в подкаталоге Chapter 7.

Применение динамических сборок

Следующий пункт в нашей программе — это различия между статическими и динамическими сборками. Статические сборки — это те сборки, с которыми мы работали все это время (и будем работать дальше): они существуют в виде файлов на жестком диске или других носителях. В отличие от них динамические сборки создаются в оперативной памяти «налету», прямо в процессе выполнения программы. Для этого используются типы, определенные в пространстве имен System.Reflection.Emit. • После того как мы создали сборку вместе с модулями и находящимися в ней типами, мы можем сохранить ее (опять-таки в процессе выполнения программы) на жестком диске. В результате будет создана новая статическая сборка! Кроме того,

при помощи типов из System.Reflection.Emit вполне возможно добавлять новые типы и члены в представление уже загруженной в оперативную память сборки.

Знакомство с пространством имен System.Reflection.Emit

Более всего типы из пространства имен System Reflection.Emit полезны для программистов, занимающихся созданием различных средств разработки. Например, представим себе, что перед нами поставлена необычная задача: разработать версию QuickBasic для работы в среде .NET (интересно, использует ли еще QuickBasic кто-нибудь из читающих эту книгу?).

При помощи System.Reflection.Emit мы можем создать средство, преобразующее код QuickBasic в динамически создаваемую сборку. Несмотря на то что такая задача выглядит несколько странной, именно так работают .NET-совместимые языки, предназначенные для использования в Web, например JScript.NET. Самые важные типы из пространства имен System.Reflection.Emit представлены в табл. 7.3.

Таблица 7.3. Некоторые типы пространства имен System.Reflection.Emit

Тип	Назначение
AssemblyBuilder	Используется для создания сборки в процессе работы программы. Этот тип может быть использован для создания как сборок типа EXE, так и DLL. Если в создаваемой сборке определен вызов метода ModuleBuilder.SetEntryPoint(), TCI будет сгенерирована сборка в формате EXE, если нет — то в формате DLL
ModuleBuilder	Для создания в процессе выполнения отдельного модуля внутри сборки
EnumBuilder TypeBuifder	Для создания типа (перечисления, класса, интерфейса и т. п.) в модуле в процессе выполнения
MethodBuilder EventBuilder LocalBuilder PropertyBuilder FieldBuilder ConstructorBuilder CustomAttributeBuilder	Эти и аналогичные типы нужны для создания внутренних членов создаваемых типов (методов, локальных переменных, свойств, конструкторов, атрибутов и т. п.) в процессе выполнения
ILGenerator	Используется для создания кода промежуточного языка (IL) для члена типа в процессе выполнения

Создаем динамическую сборку

Если вы планируете создавать «на лету» более-менее сложные сборки, вам потребуется очень хорошо знать особенности кода IL. В этой книги мы не будем подробно рассматривать код IL, однако простейшую динамическую сборку мы все же создадим — чтобы показать, как это делается. Если вам потребуется подробная информация по коду IL, мы советуем обратиться к официальной документации в разделе Tool Developers Guide (Руководство для создателей средств разработки) в .NET SDK.

В этом разделе мы создадим простую динамическую сборку. Эта сборка будет однофайловой — то есть она будет состоять из единственного модуля, который будет называться так же, как и сама сборка. В самой сборке будет определен класс

с очень редким именем — HelloWorld. В этом классе будет предусмотрен пользовательский конструктор, принимающий в качестве параметра строковое значение и присваивающий это значение переменной Msg. Кроме того, в нашем классе будет два метода. Первый, SayHello(), выводит стандартное приветствие, а второй, GetMsg(), возвращает значение переменной Msg. В общем, класс, который мы должны создавать «на лету» из другой программы, в итоге должен получиться таким:

```
// Этот класс будет создан во время выполнения при помощи типов из пространства имен
System.Reflection.Emit
public class HelloWorld
{
        private string Msg;

        // Конструктор и открытые методы класса
        HelloWorld(string s) { Msg = s; }
        public string GetMsg() { return Msg; }
        public void SayHello() { System.Console.WriteLine("Hello there"); }
}
```

Теперь наша задача — создать другой класс, который позволит создавать HelloWorld «на лету». Начнем с создания нового проекта типа Console Application и назовем этот проект DynAsmBuilder. В этом проекте создадим новый класс (пусть он называется MyAsmBuilder) с единственным методом — CreateMyAsm(), при запуске которого должны происходить создание динамической сборки с классом HelloWorld и запись ее на диск. Вот полный код этого метода (анализ будет приведен ниже):

```
// Вызывающий этот метод должен передать ему объект типа AppDomain
public int CreateMyAsm(AppDomain curAppDomain)
{
        // Определяем имя и версию создаваемой сборки
        AssemblyName assemblyName = new AssemblyName();
        assemblyName.Name = "MyAssembly";
        assemblyName.Version = new Version("1.0.0.0");

        // Создаем сборку в памяти
        AssemblyBuilder assembly = curAppDomain.DefineDynamicAssembly(assemblyName,
                                                        AssemblyBuilderAccess.Save);

        // Создаем однофайловую сборку, в которой имя единственного модуля совпадает
        // с именем самой сборки
        ModuleBuilder module = assembly.DefineDynamicModule("MyAssembly",
                                                        "MyAssembly.dll");

        // Создаем и определяем как public класс HelloWorld
        TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
                                                        TypeAttributes.Public);

        // Определяем переменную Msg (должно получиться private string Msg)
        FieldBuilder msgField = helloWorldClass.DegineField("Msg",
                        Type.GetType("System.String"), FieldAttributes.Private);

        // Определяем конструктор (как HelloWorld(string s))
        Type[] costructorArgs = new Type[1];
        constructorArgs[0] = type.GetType("System.String");
        ConstructorBuilder constructor = helloWorldClass.DefineConstructor
                (MethodAttributes.Public, CallingConventions.Standard, constructorArgs);

        ILGenerator constructorIL = constructor.GetILGenerator();
```

```
        constructorIL.Emit(OpCodes.Ldarg_0);
        Type objectClass = Type.GetType("System.Object");
        ConstructorInfo superConstructor = objectClass.GetConstructor(new Type[0]);
        constructorIL.Emit(OpCodes.Call, superConstructor);
        constructorIL.Emit(OpCodes.Ldarg_0);
        constructorIL.Emit(OpCodes.Ldarg_1);
        constructorIL.Emit(OpCodes.Stfld, msgField);
        constructorIL.Emit(OpCodes.Ret);

        // А теперь создаем метод GetMsg() (должно получиться public string GetMsg())
        MethodBuilder getMsgMethod = helloWorldClass.DefineMethod("GetMsg",
                        MethodAttributes.Public, Type.GetType("System.String"), null);

        ILGenerator methodIL = getMsgMethod.GetILGenerator();
        methodIL.Emit(OpCodes.Ldarg_0);
        methodIL.Emit(OpCodes.Ldfld, msgField);
        methodIL.Emit(OpCodes.Ret);

        // Создаем метод SayHello() (должно получиться public void SayHello()
        MethodBuilder sayHiMethod = helloWorldClass.DefineMethod("SayHello",
                                                MethodAttributes.Public, null, null);
        methodIL = sayHiMethod.GetILGenerator();
        methodIL.EmitWriteLine("Hello there!");
        methodIL.Emit(OpCodes.Ret);

        // Да будет класс HelloWorld!
        helloWorldClass.CreateType();

        // Осталось только сохранить сборку на диск
        assembly.Save("MyAssembly.dll");
        return 0;
}
```

Как мы видим, создание сборки начинается с присвоения ей имени и номера версии при помощи класса AssemblyName. Далее мы создаем сборку в памяти, используя для этого тип AppDomain (про него рассказывалось в главе 6). Обратите внимание, что вспомогательная функция CreateMyAsm() принимает объект типа AppDomain:

```
// Создаем сборку в памяти
AssemblyBuilder assembly = currAppDomain.DefineDynamicAssembly(assemblyName,
                                                AssemblyBuilderAccess.Save);
```

При вызове AppDomain.DefineDynamicAssembly() мы должны указать режим доступа — один из трех, представленных в табл. 7.4.

Таблица 7.4. Значения перечисления AssemblyBuilderAccess

Значение	Для чего оно используется
Run	Динамическая сборка должна быть запущена на выполнение, но не сохранена
RunAndSave	Запустить на выполнение и сохранить
Save	Сохранить, не запуская

Далее мы должны вставить в сборку модуль. Наша сборка состоит из единственного модуля, поэтому наша задача упрощается. Если же нам потребуется создать многофайловую динамическую сборку, нам придется использовать метод DefineDynamicModule() с еще одним параметром, представляющим имя указан-

ного модуля (например, myMod.dll). При создании однофайловой сборки имя модуля (оно же — имя двоичного файла) совпадает с именем самой сборки:

```
// Создание однофайловой динамической сборки
ModuleBuilder module = assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");
```

Используя метод ModuleBui 1 der. Defi neType(), мы можем вставить в модуль класс, структуру или интерфейс, а затем получить обратно ссылку на объект TypeBuilder, который представляет новый тип (в нашем случае это класс HelloWorld). После этого мы можем добавить в созданный класс новый член:

```
// Определяем класс "HelloWorld"
TypeBuilder helloWorldClass - module.DefineType("MyAssembly.HelloWorld",
                                                TypeAttri butes.Public);

// Определяем переменную Msg
FieldBuilder msgField = helloWorldClass.DefineField("Msg",
                        Type.GetType("System.String"), FieldAttributes.Private);
```

При создании конструктора класса нам придется прибегнуть к использованию кода IL «в сыром виде». Код IL, который помещается в реализацию конструктора при помощи метода Emit() класса ILGenerator, ответственен за присвоение принимаемого конструктором значения внутренней переменной Msg. Для метода Emit() используется перечисление Opcodes (от Operation Codes — коды операции), которое определяет допустимые команды IL. Например, OpCodes. Ret означает возврат после вызова метода, OpCodes.Stfld — собственно присвоение значения переменной и т. п. Логика создания конструктора выглядит следующим образом:

```
// Создаем конструктор класса HelloWorld
Type[] costructorArgs = new Type[1];
constructorArgs[0] = type.GetType("System.String");

ConstructorBuilder constructor =
                        helloWorldClass.DefineConstructor(MethodAttributes.Public,
                        CallingConventions.Standard, constructorArgs);

ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = Type.GetType("System.Object");

ConstructorInfo superConstructor = objectClass.GetConstructor(new Type[0]);

// Производим вызов конструктора базового класса
constructorIL.Emit(OpCodes.Call, superConstructor);

// Загружаем указатель this для объекта в стек
constructorIL.Emit(OpCodes.Ldarg_0);

// Загружаем константное 4-Сайтовое значение 0 в виртуальный стек
constructorIL.Emit(OpCodes.Ldarg_1);
// Присваиваем значение msgField
constructorIL.Emit(OpCodes.Stfld, msgField);
// Возврат из метода
constructorIL.Emit(OpCodes.Ret);
```

Для создания метода SayHello() применяется тот же самый набор средств:

```
// Создаем метод SayHello()
MethodBuilder sayHiMethod = helloWorldClass.DefineMethod("SayHello",
                                MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();

// Выводим строку на системную консоль
methodIL.EmitWriteLine("Hello there!");
methodIL.Emit(OpCodes.Ret);
```

Таким образом, мы создали метод, помеченный как public (при помощи MethodAttributes.Public), который не принимает параметров и ничего не возвращает (обратите внимание на значения null при вызове DefineMethod()). Метод EmitWriteLine(), который определен в классе ILGenerator, умеет выводить строки в стандартный канал вывода очень быстро и эффективно. Он помещен здесь как пример возможностей при использовании «голого» IL.

Как использовать динамическую сборку

Код, который позволяет создавать и сохранять нашу сборку, у нас уже есть. Нам осталось лишь обеспечить вызов этого кода. Этим будет заниматься еще один класс — AsmReader. В методе Main() будет создаваться объект класса AppDomain, который затем будет передаваться методу CreateMyAsm(). После возврата из этого метода мы применим позднее связывание для загрузки только что созданной сборки в память и вызова обоих методов класса HelloWorld. В нижеприведенном коде обратите особое внимание на то, как можно вызвать перегруженный конструктор и передать ему параметр, а также — как используется значение, возвращаемое GetMsg():

```
namespace DynAsmBuilder
{
using System;
using System.Reflection.Emit;
using System.Reflection;
using System.Threading;

public class AsmReader
{
        public static int Main(string[] args)
        {
                // Получаем объект текущего домена приложения
                AppDomain curAppDomain = Thread.GetDomain();

                // Создаем динамическую сборку
                MyAsmBuilder asmBuilder = new MyAsmBuilder();
                asmBuilder.CreateMyAsm(curAppDomain);

                // Загружаем ее
                a = Assembly.Load("MyAssembly");

                // Получаем объект класса Type для HelloWorld
                Type hello = a.GetType("MyAssembly.HelloWorld");

                // Создаем объект класса HelloWorld и вызываем перегруженный
                // конструктор
```

```
            object[] ctorArgs = new object[1];
            ctorArgs[0] = "My amazing message...";
            object obj = Activator.CreateInstance(hello, ctorArgs);

            // Вызываем метод SayHello
            MethodInfo mi = hello.GetMethod("SayHello");
            mi.Invoke(obj, null);

            // Вызываем метод GetMsg() и выводим возвращаемое им значение. Обратите
            // внимание: Invoke() возвращает объект класса Type, в котором и хранится
            // возвращаемое методом значение
            mi = hello.GetMethod("GetMsg");
            Console.WriteLine(mi.Invoke(obj, null));

            return 0;
        }
    }
}
```

Результат работы программы представлен на рис. 7.6.

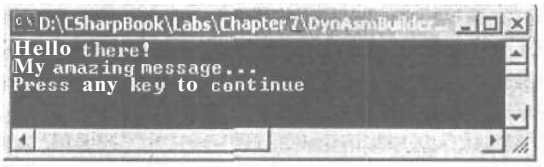


Рис. 7.6. Вызов методов динамической сборки

Как мы помним, создаваемая динамическая сборка у нас автоматически сохраняется (в каталоге проекта). Так что при желании мы можем полюбоваться на созданные нами члены этой сборки при помощи ILDasm.exe (рис. 7.7).

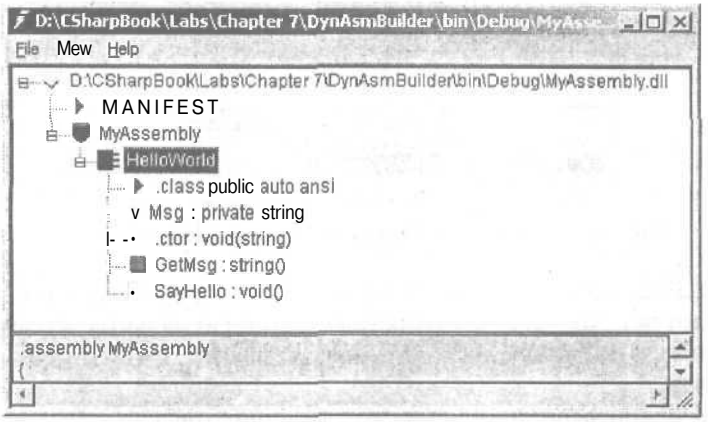


Рис. 7.7. Привет, динамическая сборка!

Код приложения DynAsmBuilder можно найти в подкаталоге Chapter 7.

Программирование с использованием атрибутов

Официальный метаязык, принятый в COM, — это IDL (Interface Definition Language, язык описания интерфейсов). IDL используется для описания набора типов внутри COM-сервера. При этом в нем активно используются атрибуты — ключевые слова IDL, заключенные в квадратные скобки. Такой «атрибутный» блок кода, заключенный в квадратные скобки, всегда относится к следующему за ним «обычному» коду. Например, когда COM-программист описывает интерфейс, он обязан использовать как минимум атрибуты [uuid] и [object]. Параметры методов помечаются атрибутами [in], [out], [in, out] и [out, retval]. Вот пример интерфейса COM, в котором используются атрибуты

```
[object, uuid(4Cb8B79A-E991-4AA4-8DB8-DD5D8751407D), oleautomation]
interface IRememberCOM : IUnknown
{
        [helpstring("If you send me a string, I will change it...")]
        HRESULT TextManipulation([in] BSTR myStr, [out, retval] BSTR* newStr);
};
```

Для метода TextManipulation() используется атрибут [helpstring], который представляет собой что-то вроде комментария к этому методу. Получать информацию о имеющихся атрибутах и их значениях можно как программным способом — непосредственно во время работы программы, так и при помощи различных средств разработки. Например, наш метод COM в утилите Object Browser из комплекта Visual Basic 6.0 выглядит так, как показано на рис. 7.8. Обратите внимание, как Object Browser показывает значение атрибута [helpstring].

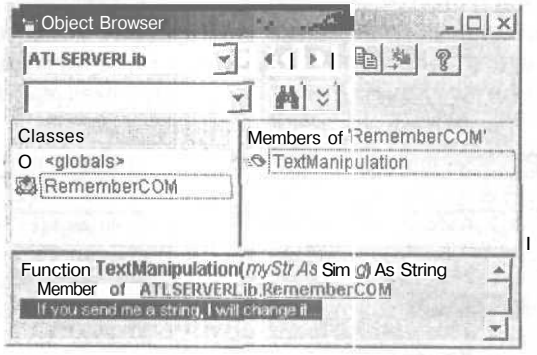


Рис. 7.8. Отображение IDL в Object Browser (Visual Basic 6.0)

Атрибуты IDL оказались настолько удачным изобретением, что они вошли в состав C# и других .NET-совместимых языков на официальных основаниях. При помощи атрибутов мы можем добавлять новую информацию в метаданные, создаваемые компилятором.

В .NET предусмотрено множество готовых атрибутов, которые можно использовать в приложениях. Кроме того, мы можем создавать свои собственные атрибуты, если мы так и не нашли подходящего из числа встроенных. Все атрибуты .NET (как встроенные, так и пользовательские) являются объектами и происходят от

одного базового класса — System.Attribute. В этом .NET сильно расходится с COM, в котором атрибуты — не более чем наборы символов.

Работа с существующими атрибутами

Атрибуты C# (как и атрибуты COM) — это аннотации, которые могут быть применены к типу (классу, интерфейсу, структуре и т. п.), члену (свойству, методу и т. д.), сборке или модулю. В разных пространствах имен .NET определено множество готовых к использованию атрибутов. Многие из них предназначены для организации взаимодействия COM/.NET, отладки и других «экзотических» аспектов создания приложений .NET. Несколько встроенных атрибутов .NET представлены в табл. 7.5.

Таблица 7.5. Некоторые встроенные атрибуты

Атрибут	Назначение
CLSCompliant	Определяет совместимость всех типов сборки с Common Language Specification (CLS). Является эквивалентом атрибута [oleautomation] в IDL
DllImport	Для вызовов традиционных файлов dll в операционной системе
StructLayout	Для определения внутреннего представления структуры
Dispid	Определяет DISPID для члена в интерфейсе COM
Serializable	Помечает класс или структуру как сериализуемые (доступные для сохранения на диске и восстановления с него)
NonSerialized	Помечает класс или структуру как несериализуемые

В качестве примера предположим, что мы хотим присвоить атрибут [Serializable] классу Motorcycle, а одному из его членов — значение [NonSerialized]. Выглядеть это может так:

```
// Этот класс можно будет сохранять на диске
[Serializable]
public class Motorcycle
{
        bool hasRadioSystem;
        bool hasHeadSet;
        bool hasSissyBar;

        // Однако при этом незачем утруждать себя сохранением этого поля
        [NonSerialized]
        float weightOfCurrentPassengers;
}
```

При помощи ILDasm.exe можно убедиться, что эти атрибуты присутствуют в определении класса (рис. 7.9).

Если нам потребуется применить к одному и тому же типу сразу несколько атрибутов, в нашем распоряжении два варианта — либо перечислить их через запятую внутри одних квадратных скобок, либо разместить их каждый в своей паре скобок друг над другом. Однако сейчас нас не интересует, кто будет использовать наши атрибуты, мы сосредоточимся на другой проблеме: как создать свой собственный, пользовательский атрибут.

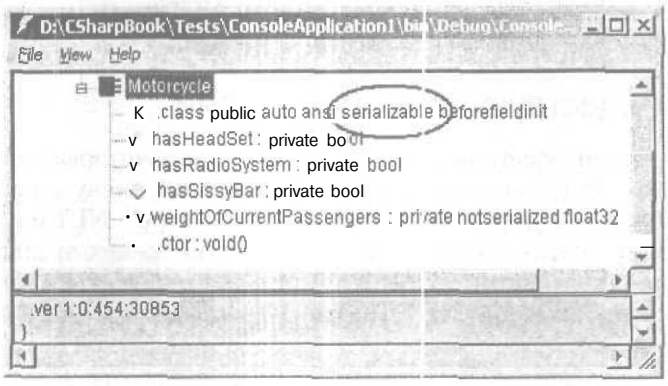


Рис. 7.9. Атрибуты представлены метаданными

Создание пользовательских атрибутов

C# (как и другие .NET-совместимые языки) позволяет создавать пользовательские атрибуты. Как мы уже говорили, все атрибуты в .NET — это классы, производные от System.Attribute. Когда мы применяем атрибут [Serializable] к классу Motorcycle, мы на самом деле применяем объект класса System.Serializable. Реально атрибут — это объект, который может быть применен к другим типам. В объектно-ориентированном программировании для описания этого подхода существует даже специальный термин — программирование, ориентированное на атрибуты (atribute-oriented programming).

Если наша цель — создать новый пользовательский атрибут, первое, что мы должны сделать — создать новый класс, производный от System.Attribute. По общепринятому соглашению об именовании имя этого класса должно оканчиваться на -Attribute. Вот пример простого атрибута, который можно будет использовать для хранения в метаданных типа текстового описания автомобиля:

```
// Пользовательский атрибут
public class VehicleDescriptionAttribute : System.Attribute
{
        private string description;
        public string Desc
        {
                get { return description; }
                set { description = value; }
        }

        public VehicleDescriptionAttribute(string desc)
        { description = desc; }
        public VehicleDescriptionAttribute(){}
}
```

Наш класс VehicleDescriptionAttribute содержит закрытую строковую переменную description, свойство Desc и перегруженный конструктор. Теперь мы можем применить этот класс к классу для автомобиля. Синтаксис, которому необходимо следовать при использовании атрибута, определяется имеющимися конструкторами атрибута:

```
// Применение пользовательского атрибута
[VehicleDescriptionAttribute("A very long, slow but feature rich auto")]
public class Winnebago
{
        public Winnebago(){}

        // Всякие члены класса
        ...
}
```

Теперь посмотрим, что получилось в результате. Как мы видим, строковое значение, передаваемое атрибуту, заключается в круглые скобки. Естественно, в определении атрибута должен существовать конструктор, который сможет принять это передаваемое значение. Теперь при помощи ILDasm.exe можно найти строковое сообщение в метаданных типов (рис. 7.10 и 7.11). Обратите внимание, что в коде IL пользовательские атрибуты помечаются ключевым словом custom.

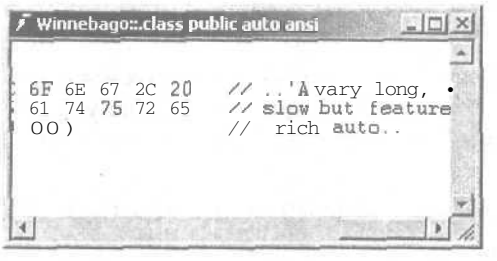


Рис. 7.10. Ваше строковое сообщение можно найти в коде IL

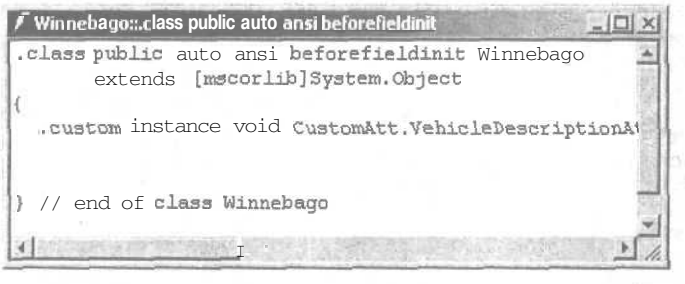


Рис. 7.11. Внутреннее представление пользовательского атрибута

Если наш пользовательский атрибут, как и положено по соглашению об именовании, заканчивается на суффикс -Atribute, этот суффикс можно опустить:

```
// Это сработает только тогда, если имя вашего пользовательского атрибута -
// VehicleDescriptionAtribute
[VehicleDescription ("A very long, slow but feature rich auto")]
public class Winnebago
{
        public Winnebago(){}

        // Всякие члены класса
        ...
}
```

Такая возможность C# не является CLS-совместимой, и другие языки .NET могут ее не поддерживать.

Как ограничить использование атрибута определенными типами

Наш пользовательский атрибут VehicleDescriptionAtribute можно использовать для самых разных целей. С точки зрения компилятора вполне допустимым будет применение атрибута не к классу, а к методу этого класса:

```
// Немного странный способ использования пользовательского атрибута, но вполне
// допустимый
public class Winnebago
{
        [VehicleDescriptionAttribute] // Вызываем конструктор по умолчанию
        public void TurnOnRadio()
        {
        }
}
```

Существует множество ситуаций, когда возможность ограничить виды типов и членов типов, к которым можно применять атрибуты, заранее определенным списком, будет очень полезна. Для этой цели можно использовать перечисление AttributeTargets:

```
// Это перечисление позволяет определить, к чему можно будет применять пользовательский
атрибут
public enum AttributeTargets
{
        All,
        Assembly,
        Class,
        Constructor,
        Delegate,
        Enum,
        Event,
        Field,
        Interface,
        Method,
        Module,
        Parameter,
        Property,
        ReturnValue,
        Struct,
}
```

Значения передаются как параметры для атрибута AttributeUsage. Этот встроенный атрибут используется компилятором C# для проверки типов, к которым можно применять данный атрибут. Первый параметр AttributeUsage — это одно или несколько приведенных выше элементов перечисления AttributeTarget, второй (необязательный) — это именованный аргумент AllowMultiple, который определяет, может ли этот аргумент применяться к одному и тому же типу несколько раз. Третий (тоже необязательный) параметр типа bool определяет, должен ли этот атрибут наследоваться производными классами.

Таким образом, если мы хотим, чтобы VehicleDescriptionAttribute мог применяться только к классам или структурам, его определение может выглядеть так;

```
// Применяем для нашего пользовательского атрибута встроенный атрибут AttributeUsage
[AttributeUsage(AttributeTargets.Class J AttributeTargets.Struct)]
public class VehicleDescriptionAttribute : System.Attribute
{
        private string description;
        public string Desc
        {
                get { return description; }
                set { description = value; }
        }

        public VehicleDescriptionAttribute(string desc)
        { description = desc; }
        public VehicleDescriptionAttribute(){}
}
```

Теперь, если попробуем применить VehicleDescriptionAttribute к методу, компилятор выдаст сообщение об ошибке (рис. 7.12),

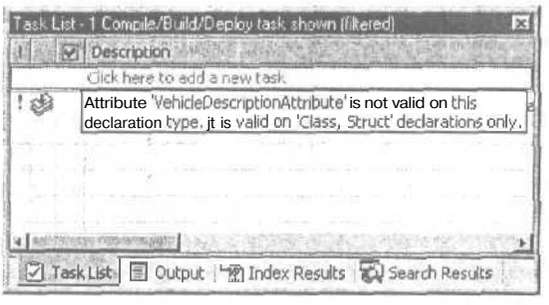


Рис. 7.12. Ограничение использования атрибута

Атрибуты уровня сборки и модуля

Вполне возможно применять атрибуты сразу ко всем типам модуля или ко всем модулям сборки. Для этого используются встроенные атрибуты [module:] и [assembly:].

Например, предположим, что мы хотим сообщить, что все типы в нашей сборке соответствуют спецификации Common Language Specification (CLS). Выглядеть это может так:

```
// Гарантируем совместимость с CLS
using System;
[assembly:System.ClsCompliantAttribute(true)]

namespace MyAttributes
{
[VehicleDescriptionAttribute("A very long, slow but feature rich auto")]
public class Winnebago
{
        public Winnebago(){}
}
}
```

Если же мы теперь попытаемся добавить код, несовместимый с CLS, например, такой:

```
// Типов ulong в CLS не предусмотрено
public class Winnebago
{
        public Winnebago(){}

        public ulong notCompliant;
}
```

то компилятор выдаст сообщение об ошибке (рис. 7.13).

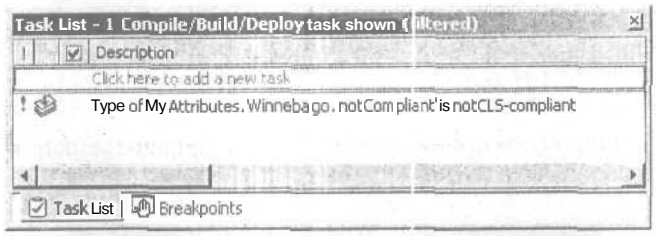


Рис. 7.13. При установленном атрибуте CLSCompliant несовместимые с CLS типы приводят к ошибке компилятора

Атрибут [CLSCompliant] в .NET является примерным эквивалентом атрибута [oleautomation] в IDL. Этот атрибут IDL гарантирует, что все члены интерфейса являются совместимыми с типом VARIANT и тем самым к ним можно будет обратиться из любого COM-совместимого языка программирования.

Атрибут [assembly:] позволяет сообщить компилятору, что атрибут [CLSCompliant] относится ко всем без исключения типам этой сборки. Таким образом, отпадает необходимость применения этого атрибута ко всем типам сборки. Еще один важный момент, на который следует обязательно обратить внимание: атрибуты [assembly:] и [module:] должны обязательно находиться за пределами каких-либо пространств имен.

Файл AssemblyInfo.cs

В любом проекте C# интегрированная среда разработки Visual Studio.NET автоматически создает файл AssemblyInfo.cs. Этот файл является местом размещения множества атрибутов уровня сборки. Наиболее важные из этих атрибутов представлены в табл. 7.6.

Таблица 7.6. Некоторые атрибуты уровня сборки

Атрибут	Назначение
AssemblyCompanyAttribute	Информация о компании
AssemblyConfiguratonAttribute	Служебная информация о назначении сборки, например retail (коммерческая версия) или debug (отладочная версия)
AssemblyCopyrightAttribute	Информация о прзвах на сборку
AssemblyDescriptionAttribute	Дружественное текстовое описание сборки
AssemblyInformationalVersionAttribute	Дополнительная информация о сборке (например, номер версии коммерческого продукта)

Атрибут	Назначение
AssemblyProductAttribute	Информация о программном продукте
AssemblyTrademarkAttribute	Информация о торговой марке
AssemblyCultureAttribute	Информация о естественном языке, поддерживаемом сборкой
AssemblyKeyFileAttribute	Информация о имени файла с парой открытый/закрытый ключ, использованной для создания цифровой подписи сборки (у сборок для общего пользования)
AssemblyKeyNameAttribute	Информация о имени контейнера с парой открытый/закрытый ключ (если такая пара вместо файла находится в контейнере CSP)
AssemblyOperationSystemAttribute	Информация об операционной системе, для работы в которой предназначена данная сборка
AssemblyProcessorAttribute	Информация о процессоре, для которого рассчитано использование данной сборки
AssemblyVersionAttribute	Определяет номер версии сборки

Как работать с атрибутами в процессе выполнения программы

И наконец, последняя тема этой главы! Как мы уже видели, атрибуты в процессе выполнения программы можно получать при помощи класса Type. Поэтому вряд ли приведенный ниже код покажется удивительным:

```
// Рефлексия для пользовательских атрибутов
public class AttReader
{
        public static int Main(string[] args)
        {
                // Получаем объект класса Type для Winnebago
                Type t = typeof(Winnebago);

                // Получаем все атрибуты данной сборки
                object[] customAtts = t.GetCustomAttributes(false);

                // Выводим информацию о каждом атрибуте
                foreach(VehicleDescriptionAttribute v in customAtts)
                        Console.WriteLine(v.Desc);

                return 0;
        }
}
```

Результат работы программы представлен на рис. 7.14.

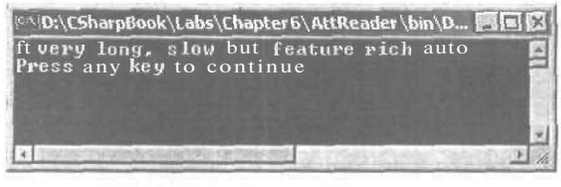


Рис. 7.14. Рефлексия для пользовательских атрибутов

Как можно догадаться, метод Type.GetCustomAttributes() возвращает массив объектов, каждому из которых соответствует атрибут. Таким образом, мы получаем список всех атрибутов, относящихся к объекту, который представлен объектом класса Type. Далее уже несложно определить, есть ли у нас определенный атрибут, каково его значение и т. п.

Код приложений CustomAtt и AttReader можно найти в подкаталоге Chapter 7.

Подведение итогов

Рефлексия — это интереснейший аспект объектно-ориентированного программирования. В мире .NET рефлексия организуется при помощи класса System.Type и типов пространства имен System.Reflection. При помощи рефлексии мы можем словно положить интересующий вас тип «под увеличительное стекло» и получить о нем всю возможную информацию — и все это программным образом, во время работы приложения.

Типы пространства имен System.Reflection.Emit позволяют создавать динамические сборки (вместе с кодом IL «в сыром виде») «на лету» — в процессе выполнения программы. Динамические сборки создаются в оперативной памяти и могут быть сохранены на диске.

Глава заканчивается рассмотрением особенностей программирования, основанного на атрибутах. При помощи встроенных и пользовательских атрибутов мы можем управлять содержимым метаданных типов и манифеста. Несмотря на то что, как правило, в большинстве случаев без пользовательских атрибутов можно обойтись, встроенные атрибуты незаменимы во многих ситуациях (например, для организации взаимодействия между сборками .NET и традиционными COM-серверами).

Окна становятся лучше: введение в Windows.Forms

8

Если вы хорошо изучили предыдущие семь глав, го к этому моменту вы обладаете солидными знаниями в области C# и всей платформы .NET. Теперь вы можете использовать приобретенные знания для создания реальных приложений.

Скорее всего, вы заинтересованы в создании приложений, которые будут обладать графическим интерфейсом пользователя (в отличие от консольных приложений, которые использовались до настоящего момента нами для тестовых целей). Эта глава поможет освоиться в пространстве имен System.Windows.Forms. Мы узнаем, как создавать главное окно приложения (пользовательский объект, производный от Form), и познакомимся с сопутствующими классами, такими как MenuItem, ToolBar, StatusBar и Application. В этой главе также будут рассмотрены вопросы, связанные с обработкой ввода пользователя (то есть событий мыши и клавиатуры) в приложениях с графическим интерфейсом.

Далее на основе изученного материала мы создадим тестовое графическое приложение, которое будет сохранять пользовательские настройки в системном реестре и записывать связанную с его применением информацию в журнал событий Window 2000. Материал этой главы позволит нам подготовиться к восприятию двух следующих глав, посвященных GDI+ и работе с элементами управления. После изучения этих трех глав мы будем вполне в состоянии создавать приложения .NET с изощренным графическим интерфейсом.

Два главных пространства имен для организации графического интерфейса

В .NET предусмотрено два главных пространства имен, обеспечивающих нас инструментами для создания приложений с графическим интерфейсом. Первое пространство имен — System.Windows.Forms — предназначено для создания обычных приложений .NET с графическим интерфейсом. Это могут быть как отдельные настольные приложения, работающие совершенно независимо, гак и клиентские части с боль-

шими возможностями (так называемые «толстые клиенты», fat clients) в распределенных клиент-серверных приложениях. Типы пространства имен Windows.Forms прячут от нас вызовы Win32 API, позволяя сосредоточиться не на технических сложностях, а на функциональных возможностях нашего приложения.

Второе пространство имен, которое также может использоваться для создания приложений с графическим интерфейсом — это System.Web.UI (и вложенное в него System.Web.UI.WebControls). Эти пространства имен используются для разработки приложений ASP.NET и позволяют создавать клиентские части приложения, работающие в любом браузере. При этом используются стандартные протоколы HTML, HTTP и прочие. Создание web-приложений и ASP.NET будет рассмотрено в главах 14 и 15 этой книги.

В настоящей главе мы рассмотрим основы построения обычных приложений .NET с графическим интерфейсом при помощи типов из пространства имен System.Windows.Forms. При этом можно заметить, что и Windows Forms, и WebForms определяют одни и те же элементы управления (например, Button или Checkbox) и используют схожие подходы. Несмотря на то что в создании приложений Windows Forms и Web Forms есть значительные различия, после того как мы освоим создание обычных графических приложений Windows, мы сможем гораздо быстрее освоить процесс создания web-приложений.

Обзор пространства имен Windows.Forms

Пространство имен System.Windows.Forms содержит огромное количество типов: классов, структур, делегатов, интерфейсов и перечислений. В этой и последующих главах мы будем подробно разбирать некоторые из этих типов. Однако вначале мы представим общий список наиболее часто встречающихся типов System.Windows.Forms (табл. 8.1).

Взаимодействие с типами Windows.Forms

Создать приложение Windows Forms можно несколькими способами. Первый — написать весь необходимый код вручную (например, в блокноте — *notepad.exe*) и откомпилировать полученный файл *cs в компиляторе C#, указав параметр /target:winexe. Надо уметь создавать приложения таким образом, поскольку это поможет нам разобраться в коде, генерируемом встроенными мастерами в Visual Studio .NET.

Второй способ — воспользоваться одним из шаблонов Windows Forms в интегрированной среде разработки Visual Studio.NET. В .NET предусмотрено множество шаблонов, мастеров и дополнительных утилит, которые при умелом использовании позволят сэкономить много времени.

Третий способ, который можно считать промежуточным между первыми двумя, — воспользоваться возможностями приложения WinDes.exe (Windows Forms Designer), поставляемого с .NET SDK (рис. 8.1). Его мы сможем найти в каталоге \Bin папки .NET SDK. Можно считать это приложение облегченной версией полной среды разработки Visual Studio.NET. При помощи его можно создавать как приложения C#, так и приложения Visual Studio.NET (есть даже возможность сохранения исходного кода в XML).

Таблица 8.1. Основные типы Windows.Forms

Класс	Назначение
Application	Этот класс представляет саму суть приложения Windows Forms. При помощи методов этого класса вы можете обрабатывать сообщения Windows, запускать и прекращать работу приложения и т. п.
ButtonBase, Button, CheckBox, ComboBox, DataGrid, GroupBox, ListBox, LinkLabel, PictureBox	Эти классы (а также многие аналогичные им) представляют элементы графического интерфейса, Они будут подробно рассмотрены в главе 10
Form	Этот тип представляет главную форму (диалоговое окно) приложения Windows Forms
ColorDialog, FileDialog, FontDialog, PrintPreviewDialog	Конечно же, в .NET предусмотрено множество готовых к употреблению диалоговых окон для выбора цветов, файлов, шрифтов и т. п. Если среди них мы так и не смогли найти ничего подходящего, мы можем создать свое собственное
Menu, MainMenu, MenuItem, ContextMenu	Эти типы предназначены для создания ниспадающих и контекстных меню
Clipboard, Help, Timer, Screen, ToolTip, Cursors	Разнообразные вспомогательные типы для организации интерактивных графических интерфейсов
StatusBar, Splitter, ToolBar, ScrollBar	Дополнительные элементы управления, размещаемые на форме

Рис. 8.1. Окно утилиты WinDes.exe

Мы с вами будем работать со средой разработки Visual Studio.NET. Однако для того, чтобы понимать код, создаваемый мастерами и помощниками Visual Studio, первые несколько приложений мы создадим вручную с самого начала.

Создание нового проекта

Чтобы лучше понимать, как создаются приложения Windows Forms, первую простую форму мы создадим сами. Итак, приступаем. Прежде всего нам нужно создать в Visual Studio.NET IDE новый *пустой* проект C#, который мы назовем MyRawWindow. Далее вставьте новое определение класса C# при помощи меню Project (Проект) ▸ Add Class (Добавить класс), как показано на рис. 8.2 (если у нас появится искушение вставить новый класс Windows Form, постараемся не поддаваться ему). Мы назовем наш новый класс MainWindow.

При создании главного окна приложения вручную нам придется как минимум использовать типы Form и Application. Оба этих типа определены в пространстве имен System.Windows.Forms, поэтому нам придется добавить ссылку на сборку System.Windows.Forms.dll. Кроме того, некоторые типы System.Windows.Forms используют типы пространства имен System, поэтому нам необходима ссылка и на System.dll. Добавим эти ссылки прямо сейчас (рис 8.3).

Создание главного окна приложения (вручную)

В мире Windows.Forms объект Form используется для представления любого окна в приложении. Если наше приложение относится к типу SDI (Single Document Interface, однодокументный интерфейс), то мы будем использовать единственное главное окно. Если же мы разрабатываем приложение MDI (Multiple Document Interface), то объект Form будет использоваться и для главного окна, и для дочерних окон. При создании нового главного окна приложения нам придется обязательно выполнить два следующих этапа:

- создать новый класс, производный от System.Windows.Forms.Form;
- настроить метод Main() таким образом, чтобы он вызывал метод Application.Run(), передавая ему созданный нами класс, производный от Form, в качестве параметра.

Таким образом, наше первое графическое приложение Windows Forms будет выглядеть следующим образом:

```
namespace MyRawWindow
{
using System;
using System.Windows.Forms;

public class MainWindow : Form
{
        public MainWindow(){}

        // Запускаем приложение
        public static int Main(string[] args)
        {
                Application.Run(new MainWindow());
                return 0;
        }
}
}
```

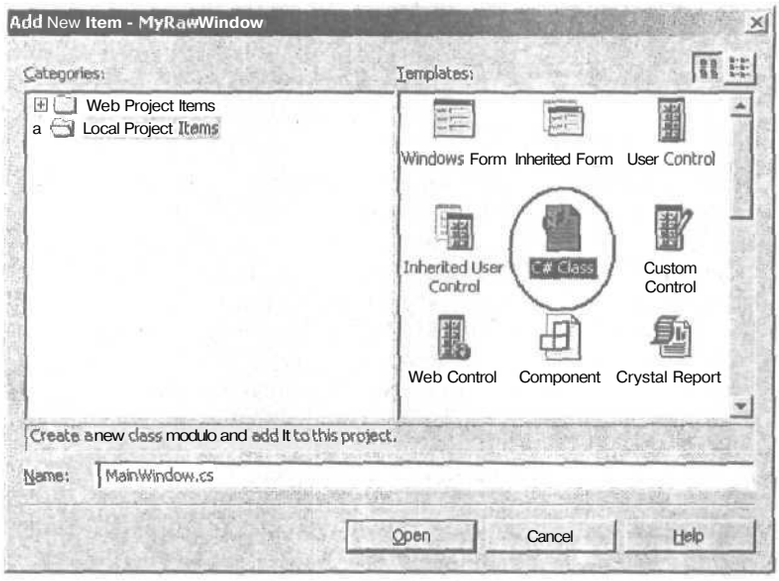


Рис. 8.2. Добавляем новый класс C#

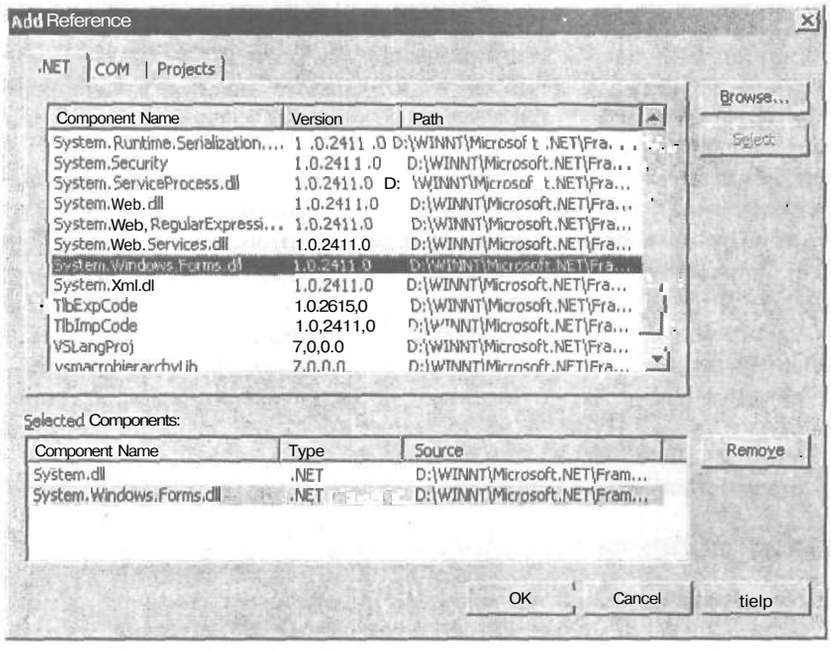


Рис. 8.3. Необходимо добавить ссылки на System.Windows.Forms.dll и на System.dll

Наше приложение должно выглядеть скромно и достойно, как показано на рис. 8.4.

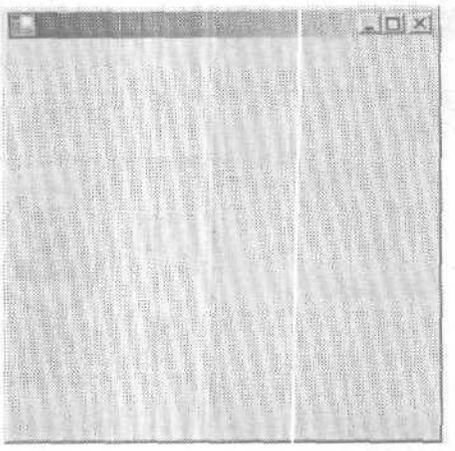


Рис. 8.4. Наша первая форма

При запуске нашего приложения на заднем плане появляется совершенно нам не нужное консольное окно. Так происходит потому, что еще не было указано, что мы создаем приложение Windows EXE. Чтобы решить эту проблему, в компиляторе командной строки csc.exe достаточно было бы указать параметр `/t:winexe`, а в IDE Visual Studio.NET нам потребуется открыть свойства проекта (можно щелкнуть правой кнопкой мыши на значке проекта в Solution Explorer и воспользоваться контекстным меню), затем открыть узел Common Properties (Общие свойства) ▸ Greneral (Общие) и выбрать для Output Type (Тип на выходе) значение Windows Application (Приложение Windows). После перекомпиляции проекта консольное окно возникать больше не будет.

Таким образом, в нашем распоряжении теперь есть главное окно приложения, которое можно минимизировать, развернуть до максимальных размеров, а потом закрыть, его размеры также можно изменять. Для этого приложения предусмотрен даже стандартный значок. То, что создавать вручную приложения с графическим интерфейсом в .NET гораздо проще, чем стандартные приложения Win32 на C/C++, заметно невооруженным глазом: разработчику традиционных приложений пришлось бы позаботиться об определении функции `WndProc`, создать точку входа в приложение `WinMain()` и возиться со структурой `WNDCLASSEX`. Однако пока наше приложение MainWindow не отличается богатством и многообразием возможностей. Мы пойдем дальше и постараемся исправить эту ситуацию.

Код приложения MyRawWindow можно найти в подкаталоге Chapter 8,

Создание проекта Windows Forms

Одно из самых важных преимуществ интегрированных средств разработки типа Visual Studio заключается в том, что многочисленные мастера и шаблоны позволяют вам сэкономить множество времени, выполняя за нас рутинную работу. Вряд ли стоит игнорировать эти возможности. Поэтому давайте закроем только что созданный нами проект и начнем работать над новым проектом C#, выбрав для него шаблон Windows Application (рис. 8.5).

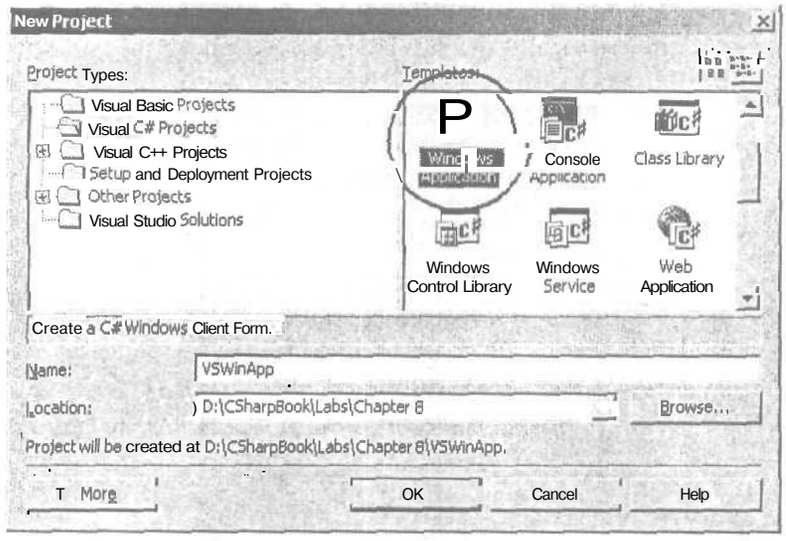


Рис. 8.5. Выбираем для проекта шаблон Windows Application

Нажав кнопку OK, чтобы подтвердить создание нового проекта, мы обнаружим, что для нас уже создан новый класс, производный от System.Windows.Forms.Form с правильно настроенным методом Main(). В свойствах проекта автоматически будут созданы ссылки на необходимые сборки библиотеки базовых классов.

Кроме того, в наше распоряжение будет предоставлен графический шаблон среды разработки (рис. 8.6), при помощи которого мы сможем «собрать» графический интерфейс приложения. При добавлении любых элементов в этот шаблон в наше приложение будет автоматически добавляться код для этих элементов (по умолчанию файл с главной формой приложения, куда будет добавляться этот код, называется Form1.cs).

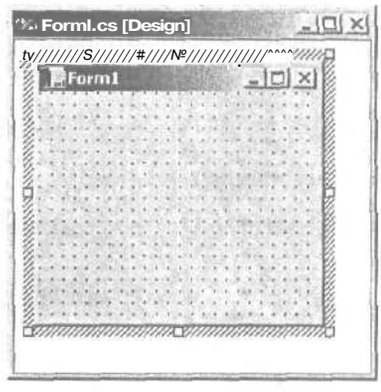


Рис. 8.6. Графический шаблон главного окна приложения

Между графическим шаблоном среды разработки и кодом приложения очень удобно перемещаться при помощи окна Solution Explorer. Чтобы перейти к просмотру кода,

достаточно просто в Solution Explorer щелкнуть правой кнопкой мыши на нужном файле и в контекстном меню выбрать View Code (Просмотреть код) — см. рис. 8.7.

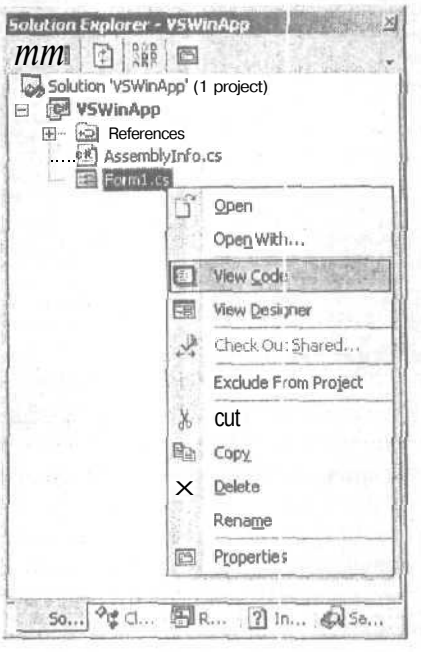


Рис. 8.7. Открываем код для формы

Можно также открыть код для формы, щелкнув дважды по любому месту на ее графическом шаблоне, но будем осторожны: таким образом можно случайно создать обработчик для событий Form Load (загрузка формы), который, возможно, нам совсем не нужен.

В *общем*, когда мы откроем код созданного при помощи шаблона Windows Application приложения, мы сможем увидеть там примерно следующее:

```
namespace VSWinApp
{
...

        public class Form1 : System.Windows.Forms.Form
        {
                private System.ComponentModel.Container components:

                public Form1()
                {
                        InitializeComponent();
                }

                public override void DisposeC)
                {
                        base.Dispose();
                        if(component$ != null)
                                components.Dispose();
                }
```

```
        #region Windows Form Designer generated code
        private void InitiializeComponenent()
        {
            this.components = new System.ComponentModel.Container();
            this.Size = new System.Drawing.Size(300,300);
            this.Text = "Form1";
        }
        #end region

        [STAhread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

Как мы можем убедиться, среда разработки делает нечто очень похожее на то, что мы делали вручную, создавая нашу первую форму. У нас есть класс, производный от System.Windows.Forms.Form, и метод Main(), который вызывает Application.Run().

Главное отличие заключается в появлении нового метода — InitializeComponent(), который окружен парой директив препроцессора (#region и #endregion). Вы можете сжать этот отрезок кода в среде разработки — тогда будет виден только комментарий "Windows Form Designer Generated Code" («код, сгенерированный Form Designer»):

```
#region Windows Form Designer generated code
private void InitiializeComponenent()
{
    this.components = new System.ComponentModel.Container();
    this.Size = new System.Drawing.Size(300,300);
    this.Text = "Form1";
}
#endregion
```

Метод InitializeComponent() обновляется Form Designer автоматически при добавлении в форму элементов управления и выполнения с ними прочих операций. Например, если мы воспользуемся окном Properties (Свойства) формы для внесения изменений в свойства Text и BackColor так, как показано на рис. 8.8, то код метода InitializeComponents() станет таким:

```
#region Windows Form Designer generated code
private void InitiializeComponenentO
{
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.BackColor = System.Drawing.Color.FromArgb(255, 128, 0);
    this.ClientSize = new System.Drawing.Size(292, 273);
    this.Text = "My Rad Form";
}
#end region
```

Класс, производный от Form, вызывает метод InitializeComponent() прямо из конструктора по умолчанию:

```
public Form1()
{
    // Required for Windows Form Designer support (Необходимо
    // для поддержки Form Designer)
    InitializeComponent();
}
```

Рис. 8.8. Окно свойств доя формы

Последняя часть кода, сгенерированная автоматически Visual Studio.NET, которая может оставаться пока непонятной, — это перегруженный метод Dispose(). Этот метод вызывается автоматически при закрытии нашей формы, и это — лучшее место для размещения кода, предназначенного для освобождения ресурсов системы. Мы поговорим о нем в ближайшем будущем более подробно, а пока скажем, что смысл этого метода заключен в следующих строках:

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

Теперь, когда мы умеем создавать главную форму нашего приложения двумя способами и знаем, что делает каждый из этих способов, настало время внимательнее посмотреть на класс Application.

Класс System.Windows.Forms.Application

Класс Application можно рассматривать как «класс низшего уровня», позволяющий нам управлять поведением приложения Windows Forms. Кроме того, этот класс определяет набор событий уровня всего приложения, например закрытие приложения или простой центрального процессора. В большинстве случаев нам не придется напрямую взаимодействовать с этим типом, однако иногда его члены могут оказаться исключительно полезными.

Наиболее важные методы этого класса (все они являются статическими) перечислены в табл. 8.2.

Таблица 8.2. Наиболее важные методы типа Application

Метод класса Application	Назначение
AddMessageFilter() RemoveMessageFilter()	Эти методы позволяют приложению перехватывать сообщения и выполнять с этими сообщениями необходимые предварительные действия. Для того чтобы добавить фильтр сообщений, необходимо указать класс, реализующий интерфейс IMessageFilter (этим мы займемся в скором времени)
DoEvents()	Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной операции. Можно сказать, что DoEvents() — это «быстрый и грязный» заменитель нормальной многопоточности
Exit()	Завершает работу приложения
ExitThred()	Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток
OLERequired()	Инициализирует библиотеки OLE. Можете считать этот метод эквивалентом .NET для вызываемого вручную метода OleInitialize()
Run()	Запускает стандартный цикл работы с сообщениями для текущего потока

Класс Application также определяет множество статических свойств, большинство из которых доступны только для чтения. Наиболее важные из них представлены в табл. 8.3. Обратите внимание, что многие из этих свойств предназначены для получения общей информации о приложении, такой как название компании, номер версии и т. п. Вопросы, связанные с получением этой информации (атрибутами сборки), должны быть нам уже знакомы по предыдущей главе.

Таблица 8.3. Наиболее важные свойства типа Application

Свойство	Назначение
CommonAppData Registry	Возвращает параметр системного реестра, который хранит общую для всех пользователей информацию о приложении
CompanyName	Возвращает имя компании
CurrentCulture	Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток
CurrentInputLanguage	Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком
ProductName	Для получения имени программного продукта, которое ассоциировано с данным приложением
ProductVersion	Позволяет получить номер версии программного продукта
StartupPath	Позволяет определить имя выполняемого файла для работающего приложения и путь к нему в операционной системе

Таким образом, при помощи многих свойств (например, CompanyName или ProductName) можно очень просто получить метаданные уровня сборки. Как мы помним из предыдущей главы, в сборке можно использовать любое количество встроенных и пользовательских атрибутов. В результате мы сможем получить значение атрибута `[assembly:AssemblyCompany("")]` при помощи свойства `Application.CompanyName` без необходимости прибегать к использованию типов, определенных в пространстве имен System.Reflection.

Таблица 8.4. События типа Application

Событие	Назначение
ApplicationExit	Возникает в тот момент, когда приложение закрывается
Idle	Возникает в тот момент, когда все текущие сообщения в очереди обработаны и приложение переходит в режим бездействия
ThreadExit	Возникает при завершении работы потока в приложении, Если работу завершает главный поток приложения, это событие возникает до события ApplicationExit

Кроме того, в типе Application определены события, представленные в табл. 8,4.

Работаем с классом Application

Чтобы проиллюстрировать возможности класса Application (и познакомиться с обработкой событий в Windows Forms), мы внесем некоторые изменения в только что созданное нами приложение с единственным пустым главным окном. Наша задача — сделать так, чтобы приложение:

- отображало при запуске некоторую информацию о самом себе;
- реагировало на событие ApplicationExit;
- выполняло некоторые предварительные действия (препроцессинг) для сообщения WM_LBUTTONDOWN.

Для начала дополним манифест нашего приложения несколькими атрибутами: именем нашего замечательного программного продукта и именем компании (про атрибуты подробно рассказывалось в предыдущей главе):

```
// Два атрибута для нашей сборки
[assembly:AssemblyCompany("Intertech, Inc.")]
[assembly:AssemblyProduct("A Better Window")]
```

Обращаться к значениям этих атрибутов мы будем прямо из конструктора нашей формы. Вначале вы получим всю необходимую нам информацию при помощи свойств типа Application, а затем выведем ее при помощи метода MessageBox.Show():

```
namespace AppClassExample
{
using System;
using System.Windows.Forms;
using System.Reflection;
public class MainForm : Form
{
        ...
        public MainForm()
        {
                GetStats();
        }

        private void GetStatsO
        {
                MessageBox.Show(Application.CompanyName, "Company:");
                MessageBox.Show(Application.ProductName, "App Name:");
```

```
            MessageBox.Show(Application.StartupPath, "I live here:");
        }
}
}
```

Теперь при запуске приложения мы получим три окна сообщения с нужной нам информацией. Последнее из этих окон представлено на рис. 8.9.

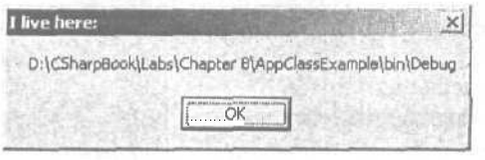


Рис. 8.9. Получение информации при помощи типа Application

Реагируем на событие ApplicationExit

Далее мы настроим нашу форму таким образом, чтобы она реагировала на событие ApplicationExit. События в приложениях Windows Forms — это обычные события C#, и реагирование на них производится точно таким же образом, как это было описано в главе 5. Таким образом, если мы хотим настроить реакцию нашего приложения на событие ApplicationExit, нам нужно создать реагирующий метод и поместить его в делегат при помощи оператора +=. Выглядеть это может так:

```
public class MainForm : Form
{
        ...
        public MainForm()
        {
                ...
                // Перехватываем событие ApplicationExit
                Application.ApplicationExit += new EventHandler(Form_OnExit);
        }

        // Метод, запускаемый обработчиком событий
        private void Form_OnExit(object sender, EventArgs evArgs)
        {
                MessageBox.Show("See ya!", "This app is dead...");
        }
}
```

Обратите внимание, что обработчик событий ApplicationExit должен соответствовать по принимаемым параметрам делегату типа System.EventHadler:

```
// Многие события приложений Windows Forms используют делегат Event Handler,
// который требует наличия двух параметров:
public delegate void EventHandler(object sender, EventArgs e);
```

У первого параметра этого делегата тип System.Object. Он представляет объект, отправляющий событие. Второй параметр, который относится к типу EventArgs, содержит информацию о самом событии. Например, если событие возникло вследствие щелчка мышью, то EventArgs будет содержать информацию, специфическую для событий мыши (например, координаты x и y указателя мыши в этот момент).

Препроцессинг сообщений при помощи класса Application

Последнее, что осталось сделать в нашем примере, — реализовать препроцессинг для сообщения WM_LBUTTONDOWN. Это стандартное сообщение Windows передается при щелчке левой кнопкой мыши на любой области формы (или элемента управления, который должен реагировать на это событие). Далее в этой главе будет представлен гораздо более простой способ реагирования на это событие, но пока нам важно разобраться в препроцессинге сообщений, поэтому мы пойдем не самым простым, но самым наглядным путем.

Если мы хотим осуществлять фильтрацию сообщений в приложении .NET, нам придется создать новый класс, реализующий интерфейс IMessageFilter. Это совсем не сложно, учитывая то, что в этом интерфейсе определен только один метод — PreFilterMessage(). Чтобы сообщение отфильтровывалось, этот метод должен возвращать true, если метод возвратит false, то реакция на сообщение будет производиться обычным порядком.

Метод PreFilterMessage() должен принимать в качестве параметра объект типа Message.Msg с числовой информацией о номере сообщения Windows (в нашем случае для WM_LBUTTONDOWN этот номер равен 513). Например:

```
// Мы должны указать это пространство имен!
using Microsoft.Win32;

// Создаем класс - фильтр сообщений
public class MyMessageFilter : IMessageFilter
{
        public bool PreFilterMessage(ref Message m)
        {
                // Перехватываем нажатие правой кнопки мыши
                if (m.Msg == 513)           // WM_LBUTTONDOWN = 513
                {
                        MessageBox.Show("WM_LBUTTONDOWN is: " + m.Msg);
                        return true;
                }
                return false;               // Все остальные сообщения игнорируются
        }
}
```

После того как новый класс, реализующий интерфейс IMessageFilter, создан, мы должны создать объект этого класса и зарегистрировать его при помощи статического метода AddMessageFilter(). Новый вариант нашего класса MainForm может выглядеть так:

```
public class MainForm : Form
{
        private MyMessageFilter msgFilter = new MyMessageFilter();
        ...
        public  MainForm()
        {
                ...
                // Добавляем (регистрируем) фильтр сообщений
                Application.AddMessageFilter(msgFilter);
        }

        // Обработчик событий
```

```
        private void Form_OnExit(object sender. EventArgs evArgs)
        {
                MessageBox.Show("See ya!", "This app is dead...");
                // Удаляем фильтр сообщений
                Application.RemoveMessageFilter(msgFilter);
        }
}
```

Теперь при щелчке левой кнопкой мыши на любом месте формы появится окно сообщения, представленное на рис. 8.10. Однако, как мы потом увидим, реализовывать таким образом фильтрацию сообщений приходится редко. Есть и более удобные способы организовывать реакцию на события. Здесь мы поместили эту конструкцию только для того, чтобы разобраться в деталях механизма работы с сообщениями Windows в .NET.

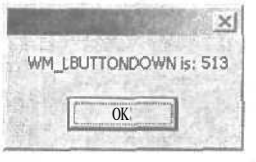


Рис. 8.10. Фильтрация сообщений

Код приложения AppClass Example можно найти в подкаталоге Chapter 8.

Анатомия формы

Мы с вами разобрались с ролью и возможностями объекта Application, и следующая наша задача — понять, как устроена сама форма, то есть класс System.Windows.Forms.Form. Каждый раз при создании нового окна приложения нам необходимо определять новый класс, производя его от System.Windows.Forms.Form. В результате создаваемый нами класс наследует от базового множество возможностей. Сам класс Form является производным от других классов в длинной цепочке наследования и собирает воедино возможности множества базовых классов. Вся цепочка наследования представлена на рис. 8.11.

Если бы мы захотели подробно описать каждый член каждого класса в этой цепочке, пришлось бы написать еще одну небольшую книгу. Однако разобраться в основных возможностях, которые передают нашему окну самые важные из базовых классов, все же необходимо.

Классы System.Object и MarshalByRefObject

Как мы уже много раз говорили, главный тип, от которого производятся любые другие типы .NET, — это тип System.Object. Поэтому Form наследует полный набор членов, определенных в System.Object (о них подробно говорилось в главе 2). Функциональность, унаследованная от MarshalByRefObject, гарантирует, что обращение к этому типу будет производиться по ссылке вместо создания локальной копии. Таким образом, к примеру, если мы обратимся к форме на удаленном компьютере, мы будем работать со ссылкой на нее, а не с локальной копией.

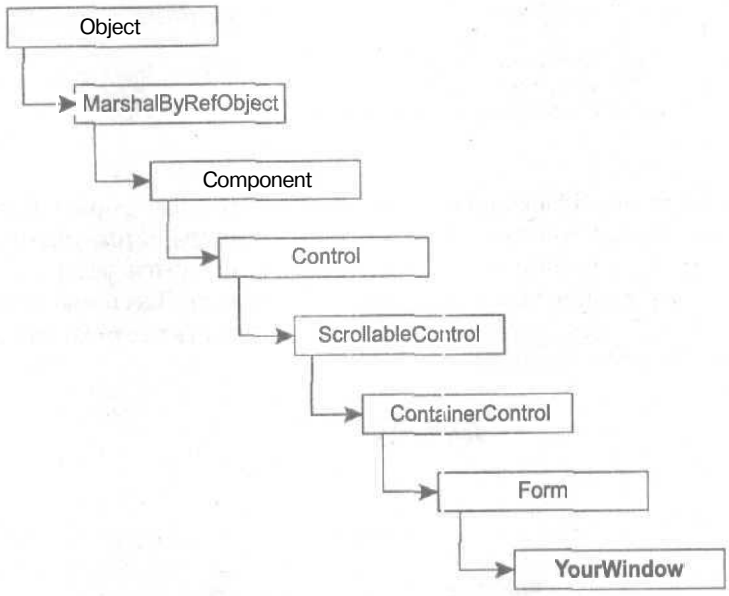


Рис. 8.11. Цепочка наследования класса Form

Класс Component

Первый базовый класс, представляющий действительный интерес, — это Component. В этом классе имеется «полуфабрикатная» реализация интерфейса IComponent. Она включает в себя свойство Site, которое возвращает еще один интерфейс — ISite. Кроме того, IComponent содержит событие Dispo:;ed:

```
public interface IComponent : IDisposable
{
        // Свойство Site
        public ISite Site { virtual get; virtual set; }

        // Событие Disposed
        public event EventHandler Disposed;
}
```

Интерфейс ISite возвращает набор методов, позволяющих элементу управления (например, кнопке — Button) взаимодействовать с контейнером, в котором этот элемент управления расположен (например, с формой):

```
public interface ISite : IServiceProvider
{
        // Свойства интерфейса ISite
        public IComponent Component { virtual get: }
        public IContainer Container { virtual get: }
        public bool DesignMode      {       virtua' get: }
        public string Name { virtual get; virtual set: }
}
```

Как правило, свойства, определяемые интерфейсом ISite, становятся объектом внимания программиста только в тех ситуациях, когда ему нужно разработать свой

собственный элемент управления. Помимо свойства Site, Component также обеспечивает реализацию метода Dispose(). Этот метод вызывается тогда, когда объект класса, производного от Component, больше не нужен. Основное его назначение — освобождать ресурсы. Например, когда окно приложения (объект, производный от Form) закрывается, метод D1spose()автоматически вызывается и для самой формы, и для всех размещенных на ней элементов управления.

Если вы знаете, как можно лучше освободить ресурсы, занятые вашим приложением, то вы вполне можете заместить метод Di spose():

```
public override void DisposeO
{
        base. DisposeO;
        // Выполняем необходимые действия
        ...
}
```

Класс Control

Следующий базовый класс, обеспечивающий вашей форме важные возможности, — это класс Control. Этот класс определяет общие черты для всех типов, относящихся к элементам графического интерфейса. Главные члены System.Windows.Form.Control позволяют настраивать размер и местонахождение окна, получать для него значение HWND, захватывать ввод с клавиатуры и мыши. Наиболее важные свойства Control представлены в табл. 8.5.

Таблица 8.5. Наиболее важные свойства класса Control

Свойство	Назначение
Top, Left, Bottom. Right, Bounds, ClientRectangle, Height, Width	Каждое из этих свойств определяет параметры какого-либо из измерений элемента управления. Bounds возвращает объект типа Rectangle (прямоугольник), который определяет размер элемента управления. ClientRectangle возвращает объект Rectangle, который соответствует размерам клиентской области элемента управления
Created, Disposed, Enabled, Focused, Visible	Каждое из этих свойств возвращает значение типа bool, определяющее текущее состояние элемента управления
Handle	Возвращает целочисленное значение — номер HWND для элемента управления
ModifierKeys	Это статическое свойство используется для проверки состояния так называемых модифицирующих клавиш (то есть Shift, Control и Alt). Оно возвращает результат в виде объекта типа Keys
MouseButtons	Это статическое свойство проверяет состояние клавиш мыши (левой, правой и средней). Оно возвращает результат в виде объекта тип MouseButtons
Parent	Возвращает объект Control, родительский по отношению к текущему
TabIndex, TabStop	Эти свойства используются для настройки последовательности перемещения с помощью клавиши Tab для элемента управления
Text	Надпись, ассоциированная с элементом управления

Базовый класс Control также определяет множество методов, которые позволяют вам взаимодействовать с типами, производными от Control, то есть с элемента-

ми управления. Некоторые наиболее часто используемые методы представлены в табл. 8.6.

Таблица 8.6. Некоторые методы класса Control

Метод	Назначение
GetStyle(), SetStyle()	Эти методы используются для установки флагов управления стилем текущего элемента управления, которое производится при помощи перечисления ControlStyles
Hide(), Show()	Эти методы управляют состоянием свойства Visible
Invalidate()	Заставляет элемент управления обновить собственное изображение путем отправки соответствующего сообщения в очередь сообщений. Этот метод перегружен таким образом, чтобы можно было обновлять не всю область, занимаемую элементом управления, а лишь ее часть
OnXXXX()	Класс Control определяет множество методов, которые могут быть замещены в производных классах для реагирования на разные события (например, OnMouseMove(), OnKeyDown(), OnResize() и т. п.). Далее в этой главе будет показано, что при организации перехвата события графического приложения возможны два основных подхода. Первый заключается в простом замещении одного из обработчиков событий. Второй — в создании пользовательского обработчика событий для выбранного пользователем делегата
Refresh()	Обновляет элемент управления и все дочерние элементы управления
SetBounds(), SetLocation(), SetClientArea()	Все эти методы используются для управления отдельными измерениями элемента управления

Настройка стиля формы

Класс Control определяет два очень интересных метода: GetStyle() и SetStyleC). Программисты, создававшие традиционные приложения WIn32, без сомнения, помнят структуру WNDCLASSEX и причудливые значения, которые приходилось присваивать многочисленным полям. В .NET все организовано гораздо проще: мы можем выбрать один из заранее готовых стилей для формы. Перечисление ControlStyles выглядит следующим образом:

```
public enum ControlStyles
{
        AllPaintingToWmPaint,
        CacheText,
        ContainerControl,
        EnableNotifyMessage,
        FixedHeight,
        FixedWidth,
        Opaque,
        ResizeRedraw,
        Selectable,
        StandardClick,
        StandardDoubleClick,
        SupportsTransparentBackColor,
        UserMouse,
        UserPaint,
}
```

Для формы можно указать сразу несколько значений этого перечисления. Кроме того, конечно же, предусмотрен и стиль по умолчанию. Рассмотрение всех по-

дробностей каждого стиля мы оставляем для самостоятельного разбора читателем (помощь можно найти в электронной документации), но некоторые моменты, связанные со стилями формы, мы все же рассмотрим.

Предположим, что на нашей форме расположен единственный элемент управления Button (кнопка). В обработчике событий для кнопки вы хотите проверить, поддерживает ли форма нужный вам стиль при помощи GetStyle():

```
// Не поддерживает!
private void btnGetStyles_Click(object sender, System.EventArgs e)
{
        MessageBox.Show(GetStyle(ControlStyles.ResizeRedraw).ToString(), "Do you have
                                                                    ResizeRedraw?");
}
```

Обеспечить поддержку формой данного стиля можно так:

```
public StyleForm()
{
        ...
        SetStyle(ControlStyles.ResizeRedraw, true);
}
```

Скорее всего, поддержка стиля ResizeRedraw для формы нам все-таки будет нужна. По умолчанию форма не поддерживает этот стиль и поэтому форма не перерисовывается автоматически при изменении ее размера. В результате может получиться очень некрасиво. Избежать этого можно двумя способами: указать поддержку формой стиля ResizeRedraw при помощи метода SetStyle() (как показано выше) или организовать перехват события Resize для формы и вызвать метод Invalidate() напрямую:

```
private void StyleForm_resize(object sender, System.EventArgs e)
{
        Invalidate();          // В результате форма будет перерисована
}
```

Обычно перехват события Resize() производится лишь тогда, когда нам нужно реагировать на изменение размера формы не только ее перерисовкой, но и выполнением каких-либо других действий.

С графической подсистемой .NET и стандартными элементами управления на форме мы познакомимся в ближайших главах. Однако чтобы проиллюстрировать эффект от установки стиля формы, мы рассмотрим очень простой пример. Предположим, что у нас в программе есть код, который обеспечивает вывод пунктирной черной линии вокруг клиентской области на форме. Если форма не будет поддерживать стиля ResizeRedraw, то при изменении размера формы может получиться безобразие (рис. 8.12).

Рис. 8.12. Форма не поддерживает стиль ResizeRedraw

Если установить значение true для флага поддержки этого стиля, то все будет в порядке (рис. 8.13).

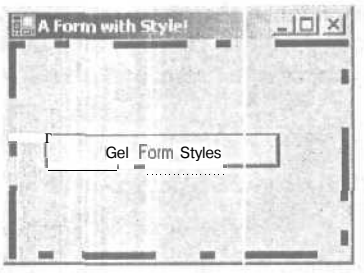


Рис. 8.13. Стиль ResizeRedraw нужен почти всегда

Код приложения FormStyles можно найти з подкаталоге Chapter 8.

События класса Control

В классе Control определен также набор событий, которые можно логически отнести к двум главным категориям: события мыши и события клавиатуры (табл. 8.7).

Таблица 8.7. Главные события класса Control

Событие	Назначение
Click, DoubleClick, MouseEnter, MouseLeave, MouseDown, MouseUp, MouseUp, MouseMove, MouseHover, MouseWheel	Все эти события предназначены для реакции на действия пользователя с мышью
KeyPress, KeyUp, KeyDown	Это — события клавиатуры

Работаем с классом Control

Конечно, в классе Control определено множество свойств, методов и событий, которые не были перечислены в предыдущих разделах. Чтобы проиллюстрировать возможности этих членов Control, мы создадим новую форму (она также будет называться MainForm), в которой:

- будут заданы конкретные размеры формы;
- будет перегружен метод Dispose();
- будет обеспечена реакция на события MouseMove и MouseUp (при помощи двух подходов);
- будет захватываться и обрабатываться ввод с клавиатуры.

Начнем с создания нового класса C#, производного от Form. Затем мы обновим конструктор этого класса по умолчанию для установки точных размеров этой формы при помощи свойств класса Control. Затем мы убедимся в том, что размеры действительно установлены, при помощи свойства Bounds и выведем текущие размеры формы в окне сообщения. Свойство Bounds возвращает тип Rectangle, опреде-

ленный в пространстве имен System.Drawing. Если мы создаем форму вручную, то нам при этом потребуется ссылка на сборку System.Drawing.dll (при использования шаблона Windows Application эта ссылка будет установлена автоматически). Таким образом, наше приложение будет выглядеть следующим образом:

```
// Необходимо для использования типа Rectangle
using System.Drawing;
...

public class MainForm : Form
{
        public static int Main(string[] args)
        {
                Application.Run(new MainFormO);
                return 0;
        }

        public MainForm()
        {
                Top - 100:
                Left = 75;
                Height = 100:
                Width = 500;
                MessageBox.Show(Bounds.ToString(), "Current rect");
        }
}
```

При запуске приложения вначале мы увидим окно сообщения (рис. 8.14).

Рис. 8.14. Используем возможности свойства Bounds

После его закрытия откроется наша форма, которая будет иметь примерно такой вид, как на рис. 8.15.

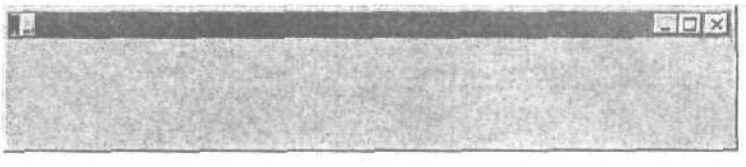


Рис. 8.15. Мы установили значения для свойств Top, Left, Height и Width

Теперь мы заместим в нашем классе унаследованный метод Component.Dispose(). Как мы уже говорили ранее, в объекте Application определено событие ApplicationExit. Если мы настроим в нашей форме перехват этого события, мы сможем точно узнать, когда наше приложение завершает свою работу, и сможем выполнить в этот момент нужные нам действия. Однако точно того же эффекта мы сможем достичь, просто прописав нужные нам действия в замещенном методе Dispose(). Дело в том, что этот

метод автоматически вызывается при закрытии формы. Обратите внимание, что при замещении этого метода очень важно не забыть вызвать Dispose() для базового класса (иначе нам придется самим реализовывать множество разных операций):

```
public class MainForm : Form
{
    ...
// Этот метод автоматически помещается в код проектов Windows Forms
    public override void Dispose()
    {
        base.Dispose();
        MessageBox.Show("Disposing this form...");
    }
}
```

Реагируем на события мыши: часть первая

Следующее, что мы должны реализовать в нашей форме, — это реакция на событие MouseUp. В нашем примере мы будем в ответ на это событие отображать координаты указателя мыши — где он находился в тот момент, когда произошло событие. Для организации реакции на событие в приложении Windows Forms у нас есть две возможности: использовать делегаты (что нам уже знакомо) и перезаписать соответствующий метод базового класса. Мы используем оба подхода, начав с обычного применения делегатов. Код обновленной MainForm может выглядеть следующим образом:

```
public class MainForm : Form
{
    public static int Main(string[] args)
    {
        Application.Run(new MainForm());
        return 0;
    }

    public MainFormO
    {
        Top - 100;
        Left = 75;
        Height = 100;
        Width = 500;
        MessageBox.Show(Bounds.ToString(), 'Current rect");

        // Перехватываем событие Mouseup
        this.MouseUp += new MouseEventHandler(OnMouseUp);
    }

    // Метод, вызываемый при возникновении события MouseUp
    public void OnMouseUp(object sender, MouseEventArgs e)
    {
        this.Text = "Clicked at: (" + e.X + ", " + e.Y +")";
    }
...
}
```

Теперь подумаем над тем, что мы сделали. Как мы помним, делегаты в приложениях Windows Forms принимают в качестве второго параметра объект класса EventArgs (или производного от него класса). Лри обработке событий мыши второй параметр — это объект производного класса MouseEventArgs. Этот тип, определенный в пространстве имен System.Windows.Forms, содержит в себе очень полезные

свойства, которые мы можем использовать для получения информации о текущем состоянии мыши и положении ее указателя (табл. 8.8).

Таблица 8.8. Свойства, определенные в классе MouseEventArgs

Свойство	Назначение
Button	Позволяет получить информацию о том, какая кнопка мыши нажата (в виде значений перечисления MouseButtons)
Clicks	Позволяет получить информацию о том, сколько раз нажата и отпущена кнопка мыши
Delta	Позволяет получить информацию (в виде положительного или отрицательного числового значения) о повороте колесика мыши
X	Позволяет получить координату X для указателя мыши во время щелчка
Y	То же самое для координаты Y

Наша реализация метода OnMouseUp() просто выводит в заголовок окна информацию о координатах указателя мыши во время щелчка. Выглядит это примерно так, как показано на рис. 8.16.

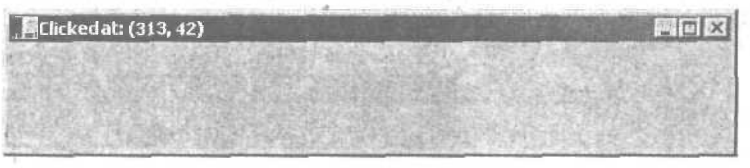


Рис. 8.16. Захват события MouseUp

Еще интереснее будет, если мы организуем захват события MouseMove и будем отображать координаты указателя непосредственно во время движения мыши над клиентской частью формы:

```
public class MainForm : Form
{
...
        public MainForm()
        {
            ...
            // Отслеживаем движения мыши (вместе с событием MouseUp
            this.MouseUp += new MouseEventHandler(OnMouseUp);
            this.MouseMove += new MouseEventHandler(OnMouseMove);
        }

        public void OnMouseUp(object sender, MouseEventArgs e)
        {
            MessageBox.Show("Stop clicking me!");
        }

        public void OnMouseMove(object sender, MouseEventArgs e)
        {
            this.Text = "Current Pos: (" + e.X + ", " + e.Y + ")";
        }
...
}
```

Какая кнопка мыши нажата?

Особенностью событий MouseUp и MouseDown является то, что они срабатывают вне зависимости от того, какая кнопка нажата (и в этом легко убедиться на примере нашего приложения). Чтобы реагировать только на нажатие определенной кнопки (вариантов три — левая, правая и средняя), необходимо организовать проверку значения свойства Button объекта класса MouseEventArgs. Диапазон возможных значений Button ограничен перечислением MouseButtons:

```
public void OnMouseUp(object sender, MouseEventArgs e)
{
        // Какая именно кнопка нажата?
        if(e.Button = = MouseButtons.Left)
                MessageBox.Show("Left click!");

        else if(e.Button = = MouseButtons.Right)
                MessageBox.Show("Right click!");

        else // Остается только средняя кнопка - MouseButtons.Middle
                MessageBox.Show("Middle click!");
}
```

Теперь при нажатии, к примеру, на среднюю кнопку мыши мы увидим окно сообщения, аналогичное представленному на рис. 8.17.

Рис. 8.17. Какая кнопка мыши нажата?

Реагируем на события мыши: часть вторая

Второй способ перехвата событий в типах, производных от Control, — заместить методы базового класса (в нашем случае OnMouseUp() и OnMouseMove()). Класс Control определяет множество виртуальных методов, определенных как protected, которые вызываются автоматически, когда происходит соответствующее событие. Если мы обновим нашу форму таким образом, чтобы для обработки событий использовался именно этот подход, вручную указывать обработчики событий нам уже не потребуется:

```
public class MainForm : Form
{
...
        public MainForm()
        {
                ...
                // Сейчас обработчики событий нам уже нужны
                // this.MouseUp += new MouseEventHandler(OnMouseUp);
                // this.MouseUp += new MouseEventHandler(OnMouseMove);
        }

        protected override void OnMouseUp(//object sender, MouseEventArgs e)
```

```
    {
        // Какая кнопка мыши нажата?
        if(e.Button = = MouseButtons.Left)
            MessageBox.Show("Left click!");
        if(e.Button = = MouseButtons.Right)
            MessageBox.Show("Right click!");
        if(e.Button - = MouseButtons.Middle)
            MessageBox.Show("Middle click!");
    }

    protected override void OnMouseMove(//object sender, MouseEventArgs e)
    {
        this.Text - "Current Pos: (" + e.X + ", " + e.Y + ")";
    }
...
}
```

Обратите внимание, что в этом случае каждый из методов принимает только один параметр, а не два, как было раньше (когда необходимо было соответствовать сигнатуре делегата MouseEventHandl er). Если мы запустим новый вариант этой программы, то никаких изменений мы не заметим (значит, все в порядке). При создании реальных приложений второй подход *с* замещением методов OnXXXX() обычно применяется только тогда, когда мы хотим выполнить перед реакцией на какое-либо событие дополнительные действия. Чаще всего используется первый подход — с применением делегатов.

Реагируем на события клавиатуры

Обработка событий клавиатуры по своим принципам практически идентична обработке событий мыши. Приведенный ниже код позволяет перехватить событие KeyUp и отобразить текстовое имя клавиши, которую мы нажали. Захват события мы будем производить при помощи делегата (если нам захочется, можно сделать то же самое, заместив метод OnKeyUp()):

```
public class MainForm : form
{
...
    public  MainForm()
    {
        Top = 100:
        Left - 75:
        Height - 100;
        Width = 500;
        MessageBox.Show(Bounds.ToString(), "Current rect");
        ...
        // Перехватываем событие KeyUp
        this.KeyUp += new KeyEventHandler(OnKeyUp);
    }

    public void OnKeyUp(object sender, KeyEventArgs e)
    {
        MessageBox.Show(e.KeyCode.ToString(), "KeyPressed!");
    }
...
}
```

Как видно из этого кода, в классе KeyEventArgs предусмотрено свойство KeyCode, при помощи которого можно получить значение из одноименного перечисления KeyCode. Как несложно догадаться, в этом перечислении хранятся названия всех клавиш на клавиатуре. Помимо свойства KeyCode, класс KeyEventArgs содержит и другие полезные свойства. Они представлены в табл, 8.9.

Таблица 8.9. Свойства класса KeyEventArgs

Свойство	Назначение
Alt	Можно получить информацию о том, нажата или нет клавиша Alt
Control	То же самое для Control
Handled	Позволяет получить или установить значение, которое говорит о том, обрабатывается ли данное событие
KeyCode	Позволяет получить код клавиши при событиях KeyDown и KeyUp
KeyData	Позволяет получить данные о нажатых клавишах при тех же событиях (отличается от KeyCode тем, что Ctrl, Alt и Shift считаются служебными, а не обычными, можно получать значения клавиатурных комбинаций с этими клавишами)
Modifiers	Позволяет получить информацию о том, какие управляющие клавиши (Ctrl, Shift и Alt) нажаты в данный момент
Shift	Можно получить информацию о том, нажата или нет клавиша Shift

Возможный результат при нажатии на одну *v* з клавиш на клавиатуре пред ставлен на рис. 8.18.

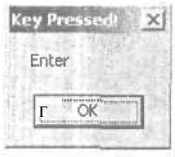


Рис. 8.18. Какая клавиша была нажата?

Код приложения Control Behaviors можно найти в подкаталоге Chapter 8.

Еще немного о классе Control

Класс Control определяет множество членов, при помощи которых мы можем настроить цвет фона и фоновые изображения, параметры шрифта, возможности перетаскивания (drag-and-drop), контекстные меню и многое другое. Кроме того, этот класс позволяет управлять местонахождением объектов классов, производных от Control (например, элементов управления). Кроме того, в этом классе предусмотрен очень важный метод OnPaint(), который позволяет управлять отображением текста, изображений, различных геометрических элементов в клиентской области формы. Наиболее важные свойства метода Control представлены в табл. 8.10.

Класс Control также определяет множество зесьма полезных методов и событий (табл. 8.11).

Таблица 8.10, Дополнительные свойства класса Control

Свойство	Назначение
AllowDrop	При установленном значении true для этого свойства разрешены операции перетаскивания (drag-and-drop) и работа с соответствующими событиями
Anchor	Определяет, какие края элемента управления будут привязаны к краям родительского контейнера
BackColor, BackgroundImage, Font, ForeColor, Cursor	Эти свойства определяют отображение клиентской области формы
ContextMenu	Определяет, какое контекстное меню будет выводиться при щелчке на форме правой кнопкой мыши
Dock	Это свойство определяет, к какому краю родительского контейнера будет «пристыкован» данный элемент управления. Например, при «стыковке» с верхним краем элемент управления будет постоянно находиться наверху контейнера
Opacity	Определяет степень прозрачности элемента управления. Допустимое значение варьируется от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный)
Region	Это свойство определяет объект Region, при помощи которого можно управлять очертаниями и границами элемента управления
RightToLeft	Это свойство предназначено для локализованных версий приложений, предназначенных для стран, где пишут справа налево

Таблица 8.11. Дополнительные методы и события класса Control

Метод или событие	Назначение
DoDragDrop() OnDragDrop() OnDragEnter() OnDragLeave() OnDragOver()	Эти методы используются в целях мониторинга операций перетаскивания для объектов классов, производных от Control
ResetFont() ResetCursor() ResetForeColor() ResetBackColor()	Эти методы используются для того, чтобы сделать значения соответствующих атрибутов пользовательского интерфейса у подчиненных элементов управления такими же, как и у соответствующего родительского контейнерного элемента управления
OnPaint()	В производных классах для организации реакции на событие Paint этот метод должен быть замещен
DragEnter DragLeave DragDrop DragOver	Эти события возникают при операциях перетаскивания
Paint	Это событие возникает в тех ситуациях, когда изображение элемента управления должно быть обновлено

Проиллюстрируем возможности этих членов. Мы изменим цвет фона для объекта Form на "Tomato" (надеюсь, вам понравится), установим прозрачность в 50 %, а курсор мыши будет изображать песочные часы. Кроме того, мы используем со-

бытие Paint для отображения текстовой строки в клиентской части формы. Вот новый код нашей формы:

```
using System;
using System.Windows.Forms;
using System.Drawing; // Для типов Color, Brush и Font

public class MainForm : Form
{
        ...
        public MainForm()
        {
                // Устанавливаем значения некоторых свойств, определенных
                // в классе Control
                BackColor = Color.Tomato;
                Opacity = 0.5d;
                this.Cursor = Cursors.WaitCursor;

                //Перехватываем событие Paint
                this.Paint += new PaintEventHandler(Form1_Paint);
        }

        private void Form1_Paint(object sender, PaintEventArgs e)
        {
                Graphics g = e.Graphics;
                g.DrawString("What a head trip...",
                        new Font("Times New Roman", 23),
                        new SolidBrush(Color.Black), 40, 10);
        }
}
```

Запустив наше приложение, мы увидим, что форма изменилась просто-таки разительно! Во-первых, она стала полупрозрачной, во-вторых, по цвету стала напоминать помидор, в-третьих, на ней появилась надпись, а указатель мыши при прохождении над ней становится совсем другим. Некоторое представление о том, что должно получиться, приведено на рис. 8.19. Обратите внимание на то, что сквозь форму просвечивают буквы кода.

Рис. 8.19. Наша форма после того, как мы поработали с некоторыми членами класса Control

Painting Basics

Событие Paint играет очень важную роль в приложениях Windows Forms. Обратите внимание, что делегат для этого события передает вызываемому методу в качестве параметра объект класса PaintEventArgs. В этом классе определены два важных свойства (табл. 8.12).

Таблица 8.12. Свойства класса PaintEventArgs

Свойство	Назначение
ClipRectangle	Позволяет получить прямоугольную область вывода изображения
Graphics	Позволяет получить объект Graphics, используемый для вывода изображения

Самое важное свойство PaintEventArgs — это Graphics, при помощи которого можно получить объект одноименного класса. Мы подробнее рассмотрим механизм работы GDI+ и подсистемы вывода изображений в целом в следующей главе, однако сейчас важно отметить, что при помощи многочисленных свойств класса Graphics мы можем выводить на область, занятую элементом управления, текстовые надписи, геометрические фигуры и изображения.

Кроме того, в нашем примере мы также воспользовались возможностями свойства Cursor для изменения формы указателя мыши. Свойству Cursor можно присвоить одно из значений перечисления Cursors (например, Arrow, Cross, UpArrow, Help и т. п.):

```
public MainForm()
{
    ...
    this.Cursor = Cursors.WaitCursor;
}
```

Код приложения MoreControlBehaviors можно найти в подкаталоге Chapter 8.

Класс ScrollableControl

В классе ScrollableControl определено всего несколько членов, главное назначение которых — обеспечить поддержку вертикальной и горизонтальной полос прокрутки. Наиболее часто используемыми типами этого класса являются свойства AutoScroll и AutoScrollMinSize. Эти свойства обеспечивают автоматическое появление полос прокрутки в тех ситуациях, когда ее содержимое не умещается в границах формы (например, если пользователь уменьшил ее размер). Свойство AutoScrollMinSize позволяет задать минимальный размер формы, при котором всегда будут появляться полосы прокрутки. Выглядеть это может так:

```
// Этот код помещается в конструктор класса или метод InitializeComponent().
// Для работы с классом Size также нужна ссылка на пространство имен System.Drawing
this.AutoScroll = true;
this.AutoScrollMinSize = new System.Drawing.Size(300, 300);
```

Обо всем остальном позаботится класс ScrollableControl. Если на форме размещено множество элементов управления, то применение полос прокрутки гарантирует, что пользователь сможет получить доступ ко всем из них. На рис. 8.20 полоса прокрутки позволяет просмотреть большой блок размещенного на форме текста.

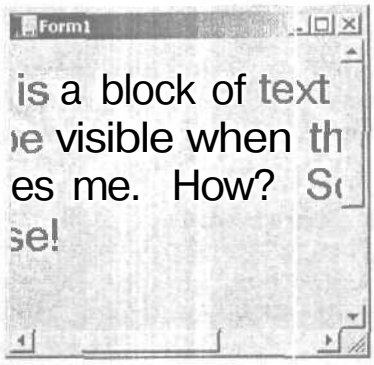


Рис. 8.20. При помощи членов класса ScrollableControl можно обеспечить автоматическое появление полос прокрутки

Класс ScrollableControl позволяет также принудительно обеспечивать появление полос прокрутки при помощи отдельных типов ScrollBar (таких как HScrollBar и VScrollBar). Информацию об остальных членах этого класса (их не так много) можно найти при помощи электронной документации по C#.

Код приложения ScrollForm можно найти в подкаталоге Chapter 8.

Класс ContainerControl

Члены класса ContainerControl позволяют управлять фокусом — то есть выделением отдельных элементов на форме (например, переход фокуса может производиться с помощью клавиши Tab). Чаще всего возможности этого класса используются в тех случаях, когда на форме присутствует множество элементов управления и мы хотим обеспечить пользователю удобство перехода но ним при помощи клавиатуры. При помощи членов этого класса мы можем программным образом получить информацию о том, какой элемент управления выделен в настоящий момент, принудительно передать фокус определенному элементу управления и т. п. Наиболее важные члены этого класса представлены в табл. 8.13.

Таблица 8.13. Члены класса ContainerControl

Член	Назначение
ActiveControl	Это свойство позволяет получать информацию о том, какой элемент управления находится в фокусе, а также помещать в фокус избранный элемент управления
ParentForm	Это свойство позволяет получать ссылку на форму-контейнер для выбранного вами элемента управления
ProcessTabKey()	Этот метод позволяет программным образом «имитировать нажатие» клавиши Tab для передачи фокуса следующему в очереди элементу управления

Кроме того, можно вспомнить, что в классе Control определены свойства TabStop и TabIndex, которые имеют непосредственное отношение к переходам между элементами управления по клавише Tab. Очень чаете эти члены используются совме-

стно с членами ContainerControl. Подробнее об использовании этих возможностей в коде программы будет говориться в главе 10, посвященной работе с элементами управления на форме.

Класс Form

Мы прошлись по всей иерархии базовых классов и подошли непосредственно к классу Form, от которого обычно и производятся формы вашего приложения. Конечно же, помимо огромного количества членов, унаследованных от Control, ScrollableControl и ContainerControl, в определении класса Form добавляется немало своих собственных членов. Наиболее важные свойства, определенные в классе Form, представлены в табл. 8.14.

Таблица 8.14. Некоторые свойства класса Form

Свойство	Назначение
AcceptButton	Позволяет получить информацию или установить кнопку на форме, которая будет активирована при нажатии пользователем на клавишу Enter
ActiveMDIChild IsMDIChild IsMDIContainer	Эти свойства предназначены для использования в контексте многооконных (MDI) приложений
AutoScale	Позволяет установить или получить значение, определяющее, будет ли форма автоматически изменять свои размеры, чтобы наилучшим образом соответствовать высоте шрифта, используемого на форме, или размерам размещенных на ней элементов управления
BorderStyle	Позволяет установить или получить стиль рамки вокруг формы. Для этого свойства используются значения перечисления FormBorderStyle
CancelButton	Позволяет установить кнопку на форме (или получить информацию о такой кнопке), которая будет автоматически активирована при нажатии пользователем на клавишу Esc
ControlBox	Позволяет установить или получить значение, определяющее, будет ли присутствовать стандартный значок (с возможностями закрытия, минимизирования и т. п.) в верхнем левом углу формы (в ее заголовке)
Menu MergedMenu	Используются для установки или получения информации о меню на форме
MaximizeBox MinimizedBox	Определяют, будут ли на форме присутствовать стандартные значки «Свернуть» и «Восстановить» в правом верхнем углу
ShowInTaskbar	Определяет, будет ли форма показываться в панели задач Windows
StartPosition	Позволяет получить или установить значение, определяющее исходное положение формы в момент выполнения программы. Могут использоваться только значения перечисления FormStartPosition
WindowState	Определяет состояние отображаемой формы при запуске. Используются значения перечисления FormWindowState

Методов, определенных непосредственно в классе Form, не так уж и много. Самые важные методы наследуются от тех базовых классов, которые мы уже рассмотрели. Некоторые методы класса Form, заслуживающие упоминания, представлены в табл. 8.15.

Таблица 8.15. Методы, определенные в классе Form

Метод	Назначение
Activate()	Активирует указанную форму и помещает ее в фокус
Close()	Закрывает форму
CenterToScreen()	Помещает форму в центр экрана
LayoutMDI()	Размещает все дочерние формы на родительской в соответствии со значениями перечисления LayoutMDI
OnResize()	Может быть замещен для реагирования на событие Resize()
ShowDialog()	Отображает форму как диалоговое окно (подробнее о диалоговых окнах рассказывается дальше в этой главе)

Кроме того, класс Form определяет набор событий. Наиболее важные из этих событий представлены в табл. 8.16.

Таблица 8.16. Некоторые события класса Form

Событие	Назначение
Activate	Происходит при активизации формы (когда она выходит в активном приложении на передний план)
Closed, Closing	Происходят во время закрытия формы
MDIChildActive	Возникает при активизации дочернего окна

Используем возможности класса Form

К этому моменту вы уже наверняка вполне освоились с возможностями, которыми обладают класс Form и производные от него классы, используемые в приложениях Windows Forms. Ниже представлен код приложения, в котором используются самые разные члены классов, участвующих в цепочке наследования Form:

```
public class MainForm: Form

{
    ...
    public MainForm()
    {
        // Настраиваем исходный облик нашей формы
        BackColor = Color.LemonChiffon;        // Цвет фона:
        Text = "My Fantastic Form";            // Заголовок формы:
        Size - new Size(2QO, 200);             // Размер 200*200:
        CenterToScreen();                      // Помещаем форму в центр экрана:

        // Перехватываем события
        this.Resize += new EventHandler(this.MainForm_Resize);
        this.Paint += new PaintEventHandler(this.MainForm_Paint);
    }

    private void MainForm_Resize(object sender. System.EventArgs e)
    {
```

```
            // При изменении размера формы ее изображение нужно обновить! Можно также
            // вместо вызова Invalidate просто установить поддержку стиля ResizeRedraw
            Invalidate();
        }

        // Выводим на форму текстовую строку. Для типов, используемых в этом методе,
        // нам потребуется ссылка на System.Drawing
        private void MainForm_paint(object sender, PaintEventArgs e)
        {
            Graphics g * e.Graphics;
            g.DrawString("Windows Forms is for building GUIs!",
                new Font("Times New Roman", 20),
                new SolidBrush(Color.Black),
                this.DisplayRectangle);  // Выводим в клиентской прямоугольной
                                         //  области
        }
    }
```

Таким образом, форма с указанными нами размерами и цветом фона начнет свое существование точно по центру экрана. При изменении ее размеров (то есть при наступлении события Resize()) форма будет перерисована (будет вызван метод Invalidate()). Кроме того, на форме будет выведена текстовая надпись.

Код приложения SimpleFormApp можно найти в подкаталоге Chapter 8.

Создаем меню

Теперь, когда мы познакомились со строением форм Windows в .NET, следующая наша задача — научиться создавать на формах систему меню, при помощи которой пользователь сможет выполнять различные операции. В пространстве имен System.Windows.Forms предусмотрено большое количество типов для организации ниспадающих главных меню (расположенных в верхней части формы) и контекстных меню, открывающихся по щелчку правой кнопки мыши. Мы начнем с создания стандартного ниспадающего меню, которое позволит пользователю выйти из приложения, выбрав пункт File (Файл) ▸ Exit (Выход). То, что должно получиться, представлено на рис. 8.21.

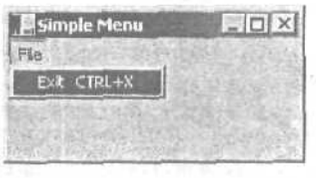


Рис. 8.21. Простое меню на форме

Первый класс, с которым мы познакомимся, — это класс System.Windows.Forms.Menu. Этот класс является базовым для таких часто используемых производных классов, как MainMenu, MenuItem и ContextMenu, которые мы рассмотрим чуть позже. System.Windows.Forms.Menu — это абстрактный класс, и мы не сможем создать объект этого класса напрямую. В приложении используются только объекты производных типов. Класс Menu обеспечивает важнейшие функции любых видов меню —

возможность доступа к элементам меню, слияние меню в приложениях MDI и т. п. Иерархия основных классов Windows.Forms, предназначенных для создания меню в приложениях, представлена на рис. 8.22.

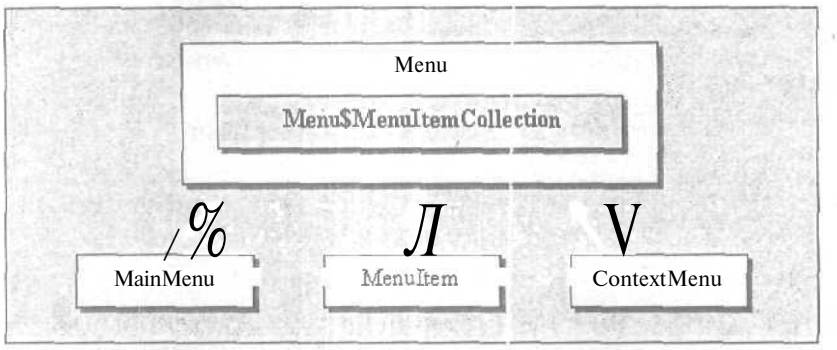


Рис. 8.22. Иерархия классов Windows.Forms для работы с меню

Обратите внимание, что класс Menu определяет вложенный класс MenuItemCollection. Этот вложенный класс наследуется всеми тремя производными классами; и MainMenu, и MenuItem, и ContextMenu. Как, наверное, вы уже догадались по названию, MenuItemCollection предназначен для хранения информации об элементах меню, к которым можно получить доступ через свойство Menu.MenuItems (о нем будет сказано чуть позже). Главные члены базового класса Menu представлены в табл. 8.17.

Таблица 8.17. Члены класса Menu

Член	Назначение
Handle	Это свойство обеспечивает доступ к значению HMENU, идентифицирующему данное меню
IsParent	Это свойство определяет, содержит ли данное меню какие-либо подменю или оно является конечным
MdiListItem	Это свойство возвращает объект MenuItem, который содержит список дочерних окон приложения MDI
MenuItems	Это свойство возвращает объект вложенного класса Menu.MenuItemCollection, который представляет подменю, являющееся вложенным для текущего меню
GetMainMenu()	Возвращает объект MainMenu, в котором содержится текущее меню
MergeMenu()	Объединяет элементы меню в единое меню согласно данным, содержащимся в свойствах mergeType и mergeOrder. Используется для слияния меню контейнера в приложении MDI с меню дочернего окна
CloneMenu()	Создает меню, являющееся полной копией другого меню (создается полная локальная копия, а не ссылка на существующее меню)

Вложенный класс Menu$MenuItemCollection

Возможно, самое важное свойство класса Menu — это свойство MenuItems, возвращающее объект вложенного класса Menu$MenuItemCollection. Этот вложенный класс представляет набор всех подменю для текущего меню — то есть для объекта класса, производного от Menu.

Например, если мы создали объект класса MainMenu, представляющий меню верхнего уровня File (Файл), то в коллекцию MenuItemCollection вы можете поместить объекты класса MenuItems, такие как Open (Открыть), Save (Сохранить), Close (Закрыть), Save As (Сохранить как) и т. п. Конечно, в MenuItemCollection предусмотрены члены, которые позволяют без каких-либо проблем добавлять и удалять объекты класса MenuItem, получать о них информацию, а также обращаться к конкретному элементу данной коллекции. Некоторые наиболее важные члены MenuItemCollection представлены в табл. 8.18.

Таблица 8.18. Члены вложенного типа Menu$MenuItemCollection

Член	Назначение
Count	Возвращает текущее количество объектов класса MenuItem в коллекции
Add()	Добавляет новый объект класса MenuItem в коллекцию. Существует множество перегруженных вариантов этого метода, которые позволяют указывать клавиши быстрого доступа, делегаты и многое другое
Remove()	Для удаления объектов класса MenuItem из коллекции
AddRange()	Позволяет добавить в коллекцию за один раз массив объектов MenuItem
Clear()	Удаляет все объекты MenuItem из коллекции
Contains()	Используется для того, чтобы определить, присутствует ли определенный объект класса MenuItem в коллекции

Создание системы меню в приложении

Теперь, когда мы выяснили возможности абстрактного класса Menu и вложенного в него класса MenuItemCollection, можно приступать к созданию нашего простого меню File (Файл). Первое, что мы должны сделать — создать объект MainMenu. Этот класс представляет элементы меню верхнего уровня (такие как File (Файл), Edit (Правка), View (Вид), Tools (Сервис), Help (Справка) л т. п.) Код для этого может выглядеть следующим образом:

```
public class MainForm : Form
{
        // Главное меню для нашей формы
        private MainMenu mainMenu;

        public MainForm()
        {
                // Создаем это главное меню
                mainMemu = new MainMenu();
        }
```

Г

После того как мы создали объект MainMenu, наша следующая задача — использовать метод Menu$MenuItemCollection.Add(), чтобы вставить элемент меню верхнего уровня (в нашем случае File). Метод Menu$MenuItemCollection.Add() вернет новый объект MenuItem, который представляет только что вставленный в главное меню элемент File.

Чтобы добавить в меню File пункт Exit (Выход), необходимо вставить в коллекцию Menu$MenuItemCollection, которая находится внутри объекта MenuItem (конечно,

представляющего в нашем случае меню File), новы и объект MenuItem. Если потребуется поместить внутрь меню прочие элементы, это делается точно так же. После того как вставка элементов в меню завершена, последнее, что мы должны сделать, — присоединить созданную систему меню к нашей форме, используя для этого свойство Form.Menu. Выглядеть все это может так:

```
public class MainForm: Form
{
        // Главное меню для Form
        private MainMenu mainMenu;

        public MainForm()
        {
                // Создаем главное меню
                mainMenu = new MainMenu();

                // Создаем меню File и добавляем его в MenuItemCollection
                MenuItem miFile = mainMenu.MenuItems.Add("&File");

                // Теперь создаем подменю Exit и добачляем его в меню File. Этот вариант
                // Add() принимает: (1) создаваемый объект MenuItem; C2) создаваемый
                // делегат (EventHandler); (3) необязательную клавиатурную комбинацию
                // быстрого доступа
                miFileMenuItems.Add(new MenuItem("E&xit",
                                new EventHandler(this.FileExit_Clicked), Shortcut.CtrlX));

                // Присоединяем главное неню к объекту Form
                this.Menu = mainMenu:
        }
        ...
}
```

Обратите внимание на использование символа амперсанда (&) в названии элемента меню. Символ, перед которым стоит амперсанд, определяет букву, которая будет подчеркнута в названии элемента меню. Данный элемент будет активизироваться при нажатии соответствующей этой букве клавиши. Таким образом, если мы указали название меню как &File, мы можем открыть меню File при помощи клавиатурной комбинации Alt+F.

При добавлении пункта меню Exit мы указали еще и дополнительную клавиатурную комбинацию Ctrl+X. Значения для подобных клавиатурных комбинаций приведены в перечислении System.Windows.Forms.Shortcut. В нем представлены как привычные всем комбинации Ctrl+C, Ctrl+V, F1, F2 INS, так и более экзотические.

Теперь мы можем создать уже полное приложение, использующее меню. Обратите внимание на небольшой трюк — как, оказывается, можно установить, например, значение свойства BackColor при помощи метода MainMenu.GetForm():

```
// Простое приложение с главным меню
public class MainForm : Form
{
        // Главное неню для формы
        private MainMenu mainMenu;

        // Запускаем приложение
        [STAThread]
        public static int Main(string[] args)
```

```
{
        Application.Run(new MainForm());
}

// Создаем форму
public MainForm()
{
        // Настраиваем исходный облик и местонахождение формы
        Text - "Simple Menu";
        CenterToScreen();

        // Создаем объект главного меню
        mainMenu = new MainMenu();

        // Создаем меню File | Exit
        MenuItem miFile - mainMenu.MenuItems.Add("&File");
        miFile.MenuItems.Add(new MenuItem("E&xit",
                new EventHandler(this.FileExit_Clicked), Shortcut.CtrlX));

        // Присоединяем главное меню к объекту Form
        this.Menu = mainMenu;

        // MainMenu.GetForm() возвращает ссылку на форму, на которой расположено
        // меню. Поэтому мы можем сделать такой маленький фокус:
        mainMenu.GetForm().BackColor * Color.Black;
}

// И не забыть про обработчик событий для File ▸ Exit
private void FileExit_Clicked(object sender, EventArgs e)
{
        this.Close();       // Выход из приложения
}
}
```

Добавляем еще одно меню верхнего уровня

Предположим, что техническое задание на наш проект изменилось: теперь кроме меню File ▸ Exit нам необходимо создать еще одно меню верхнего уровня Help (Справка) с единственным пунктом About (0 программе). Выглядеть все это должно так, как представлено на рис. 8.23.

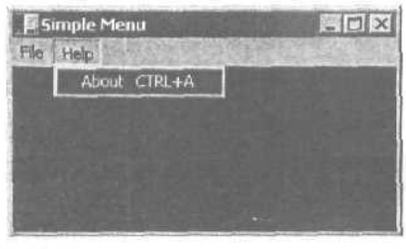


Рис. 8,23. Появляется новое меню

Логика при создании Help ▸ About, конечно, будет точно такой же, как и при создании File ▸ Exit мы просто добавляем новый объект MenuItem в коллекцию внутри объекта MainMenu, а затем добавляем в этот объект MenuItem еще один MenuItem для About:

```
public class MainForm : Form
{
        private MainMenu mainMenu;
...
        public MainForm()
        {
                // Создаем меню File ▸ Exit
                MenuItem miFile = mainMenu.MenuItems.Add("&File");
                miFile.MenuItems.Add(new MenuItem("E&xit",
                                new EventHandler(this.FileExit_Clicked), Shortcut.CtrlX));

                // А теперь - еще и меню Help ▸ About
                MenuItem miHelp = mainMenu.MenuItems.Add("Help");
                miHelp.MenuItems.Add(new MenuItem("&About",
                                new EventHandler(this.HelpAbout_Clicked), Shortcut.CtrlA));
                ...
        }
        // И обработчик события для Help ▸ About
        private void HelpAbout_Clicked(object sender, EventArgs e)
        {
                MessageBox.Show("The amazing menu app...");
        }
}
```

Код приложения SimpleMenu можно найти в подкаталоге Chapter 8.

Создаем контекстное меню

Создавать обычные ниспадающие меню мы уже научились. Однако в приложениях часто используется и другой вид меню — контекстные (pop-up), которые открываются по щелчку правой кнопкой мыши. Контекстные меню создаются при помощи класса ContextMenu. Как и в случае с MainMenu, наша задача — добавить объекты MenuItem в коллекцию MenuItemCollection внутри объекта ContextMenu. В приведенном ниже примере контекстное меню позволяет пользователю выбрать размер шрифта для текстовой строки, выводимой на форму:

```
namespace MainForm
{
        // Вспомогательная структура для установки размера шрифта
        internal struct TheFontSize
        {
                public static int Huge = 30;
                public static int Normal = 20;
                public static int Tiny = 8;
        }
        public class MainForm : Form
        {
                // Исходный размер шрифта
                private int currFontSize = TheFontSize.Normal;

                // Контекстное меню формы
                private ContextMenu popUpMenu;

                public static void Main(string[] args)
                {
                        Application.Run(new MainForm());
                }
```

```
        private void MainForm_Resize(object sender. System.EventArgs e)
        {
                Invalidate();
        }

        public MainForm()
        {
                // Прежде всего создаем контекстное меню
                popUpMenu = new ContextMenu();

                // Теперь добавляем в контекстное неню элементы
                popUpMenu.MenuItems.Add("Huge", new EventHandler(PopUp_Clicked));
                popUpMenu.MenuItems.Add("Normal", new EventHandler(PopUp_Clicked));
                popUpMenu.MenuItems.Add("Tiny", new EventHandler(PopUp_Clicked));

                // Теперь подключаем контекстное неню к форме
                this.ContextMenu = popUpMenu;

                // Ставим обработчики событий
                this.Resize += new System.EventHandler(this.MainForm_Resize);
                this.Paint += new PaintEventHandler(this.MainForm_Paint);
        }

        // Обработчик для PopUp_Clicked (всех трех пунктов)
        private void Popup Clicked(object sender. EventArgs e)
        {
                // Ориентируемся на строковое имя выбранного пользователем элемента
                // неню
                MenuItem miClicked = (MenuItem)sender;
                string item = miClicked.Text;

                if(item = = "Huge")
                        currFontSize = TheFontSize.Huge;
                if(item = - "Normal")
                        currFontSize = TheFontSi ze. Normal;
                if(item = = "Tiny")
                        currFontSize = TheFontSize.Tiny;
                Invalidate();
        }

        private void MainForm_Paint(object sender, PaintEventArgs e)
        {
                Graphics g = e.Graphics;
                g.DrawString("Please click on me...",
                        new Font("Times New Roman", (float)currFontSize),
                        new SolidBrush(Color.Black),
                        thi s.Di splayRectangle);
        }
    }
}
```

Обратите внимание, что для всех элементов контекстного меню мы назначили один-единственный общий для всех обработчик событий. При выборе пользователем любого пункта в контекстном меню запускается один и тот же метод PopUp_Cl icked(). Но, конечно, нам все-таки надо реагировать на разные пункты меню по-разному. Для этого мы используем значение свойства Text передаваемого обработчиком событий объекта sender (предварительно приведенного к типу Menultem), Далее в зависимости от значения этого свойства выбирается нужное действие. Все

работает просто замечательно, хотя, пожалуй, создание версии приложения с интерфейсом на другом языке потребует дополнительных усилий:

```
// Обработчик для PopUp_Clicked (всех трех пунктов)
private void PopUp_Clicked(object sender. EventArgs e)
{
        // Ориентируемся на строковое имя выбранного пользователем элемента меню
        MenuItem miClicked - (MenuItem)sender;
        string item = miClicked.Text;

        if(item - = "Huge")
                currFontSize = TheFontSize.Huge;
        if(item = = "Normal")
                currFontSize - TheFontSize.Normal;
        if(item = = "Tiny")
                currFontSize = TheFontSize.Tiny;

        // А это — чтобы перерисовать форму с новый шрифтом для текста на ней:
        Invalidate();
}
```

Еще один момент, на который стоит обратить внимание. После создания объекта ContextMenu мы подключаем его к форме при помощи свойства ContextMenu, определенного в классе Control. Это значит, что контекстное меню может быть у всех объектов, производных от класса Control, то есть у всех элементов управления. Например, если нам этого хочется, отдельное контекстное меню может быть у любой кнопки в диалоговом окне. Появляться это контекстное меню будет только при щелчке правой кнопкой мыши на области, занимаемой данной кнопкой.

Дополнительные возможности меню

В классе Menultem определены свойства, при помощи которых можно, к примеру, устанавливать флажок напротив пункта меню, прятать пункты меню, делать некоторые пункты меню недоступными и т. п. Вот перечень свойств, обеспечивающих подобные возможности (табл. 8.19).

Таблица 8.19. Свойства Menultem, обеспечивающие дополнительные возможности меню

Свойство	Назначение
Checked	Позволяет получить или установить значение, определяющее, будет ли установлен флажок рядом с текстом пункта меню
DefaultItem	Позволяет получить или установить значение, определяющее, какой пункт меню выбран по умолчанию
Enabled	Получает или устанавливает значение, определяющее, будет ли доступен тот или иной пункт меню
Index	Позволяет получить или установить значение, определяющее позицию пункта меню
MergeOrder	То же самое для позиции пункта при слиянии двух меню
MergeType	Позволяет получить или установить значение, определяющее поведение пункта меню при слиянии этого меню с другим
OwnerDraw	Позволяет получить или установить значение, определяющее, кто будет ответствен за прорисовку пункта меню — Windows или ваш специально для этого предназначенный код

Свойство	Назначение
RadioCheck	Позволяет получить или установить значение, которое определяет, будет ли рядом с названием пункта меню показываться переключатель (вместо флажка при использовании Checked)
Shortcut	Позволяет получить или установить клавиатурную комбинацию, используемую для активизации элемента меню в приложении
ShowShortcut	Позволяет получить или установить значение, определяющее, будет ли такая клавиатурная комбинация быстрого доступа выведена рядом с названием пункта меню
Text	Позволяет получить или установить название пункта меню

Для иллюстрации этих возможностей мы изменим наше приложение с контекстным меню таким образом, чтобы рядом с пунктом меню, соответствующим выбранному в настоящий момент размером шрифта, ставился флажок. Сам флажок установить совершенно не сложно: для этого достаточно просто установить для свойства Checked пункта меню значение true. Однако для того, чтобы правильно устанавливать флажок напротив нужного пункта меню, нам потребуется дополнительная логика. Один из возможных подходов заключается в том, чтобы определить четыре ссылки на объекты MenuItem: три будут соответствовать варианту «флажок установлен» для каждого из наших трех пунктов меню, а четвертая (которая в списке идет первой) будет представлять выбранный в настоящий момент элемент меню:

```
public class MainForm : Form
{
        // Текущий размер шрифта
        private int currFontSize - TheFontSize.Normal;

        // Контекстное меню для формы
        private ContextMenu popUpMenu;

        // Дополнительные ссылки для отслеживания пункта меню, напротив которого надо
        // установить флажок
        private Menultem currentCheckedItem;   // Представляет выбранный в настоящий
                                               // момент пункт меню
        private Menultem checkedHuge;
        private Menultem checkedNormal;
        private MenuItem checkedTiny;
...
}
```

Следующая задача — связать каждую из этих ссылок с нужным пунктом меню в нашем приложении. Для этого нам потребуется внести следующие изменения в конструктор формы:

```
// Конструктор формы
public MainForm()
{
        // Настраиваем исходный облик формы
        Text = "PopUp Menu";
        CenterToScreen();

        // Прежде всего создаем контекстное меню
        popUpMenu = new ContextMenu();
```

```
        // А теперь добавляем пункты меню
        popUpMenu.MenuItems.Add("Huge", new EventHandler(PopUp_Clicked));
        popUpMenu.MenuItems.Add("Normal", new EventHandler(PopUp_Clicked));
        popUpMenu.MenuItems.Add("Tiny", new EventHandler(PopUp_Clicked));

        this.ContextMenu = popUpMenu;

        // Теперь привязываем каждую из наших ссылок на MenuItem к элементам контекстного
        // меню
        checkedHuge = this.ContextMenu.MenuItems[0];
        checkedNormal = this.ContextMenu.MenuItems[1];
        checkedTiny - this.ContextMenu.MenuItems[2];

        // А теперь ставим флажок напротив Normal:
        currentCheckedItem = checkedNormal;
        currentCheckedItem.Checked - true;
}
```

Теперь у нас все готово для отслеживания выбранного пользователем пункта меню и даже предусмотрена изначальная установка флажка напротив Normal. Последнее, что осталось сделать, — внести изменения в обработчик событий PopUp_Clicked() таким образом, чтобы при выборе пользователем пункта меню напротив этого пункта меню еще и устанавливался флажок:

```
private void PopUp_Clicked(object sender. EventArgs e)
{
        // Снимаем флажок с выбранного в настоящий момент пункта
        currentCheckedItem.Checked = false;

        // Определяем название выбранного пользователем пункта меню
        MenuItem miClicked = (MenuItem)sender;
        string item = miClicked.Text;

        // В ответ на выбор пользователя устанавливаем нужный размер шрифта и флажок
        // напротив пункта меню
        if(item = = "Huge")
        {
                currFontSize = TheFontSize.Huge;
                currentCheckedItem = checkedHuge:
        }
        if(item = = "Normal")
        {
                currFontSize = TheFontSize.Normal;
                currentCheckedItem = checkedNormal;
        }
        if(item = = "Tiny")
        {
                currFontSize * TheFontSize.Tiny;
                currentCheckedItem = checkedTlny;
        }

        // А теперь устанавливаем флажок
        currentCheckedItem.Checked = true;
        Invalidate();
}
```

То, что должно получиться в итоге, представлено на рис. 8.24.

Код приложения PopUpMenu можно найти в подкаталоге Chapter 8.

Рис. 8.24. Ставим флажок напротив пунктов меню

Создаем меню при помощи IDE Visual Sudio.NET

Знание — сила. Именно поэтому мы создавали наши меню вручную. Однако, когда мы уже знаем, что такое меню в C# и как работать с относящимся к нему кодом, вполне можно сэкономить время и создать меню при помощи графических средств, предлагаемых Visual Studio.NET.

Первое, что нам нужно сделать — создать новый проект Windows Application. Далее в окне Toolbox дважды щелкнем на значке MainMenu (рис. 8.25).

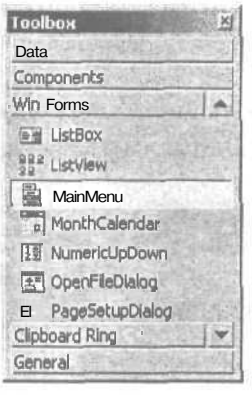


Рис. 8.25. Добавляем меню при помощи графических средств Visual Studio

Шаблон формы изменится: на самой форме появится меню с предложением ввести название первого элемента, а внизу — символ для объекта главного меню. Введем название первого пункта меню, чтобы ввести названия следующих пунктов, дважды щелкнем на этой же области! Все это выглядит так, как представлено на рис. 8.26.

При помощи графических средств мы можем также настроить свойства любого элемента меню, а также — обработчик событий (рис. 8.27).

После того как в строке, выделенной на рис. 8.27, мы введем имя обработчика событий, Visual Studio автоматически перейдет в окно кода, в котором нам будет предложено создать логику для сгенерированного Visual Studio метода (в нашем случае OnMenuSave()):

```
protected void OnMenuSave (object sender. System EventArgs e)
{
        // Здесь мы определяем реакцию на выбор пользователем пункта меню
}
```

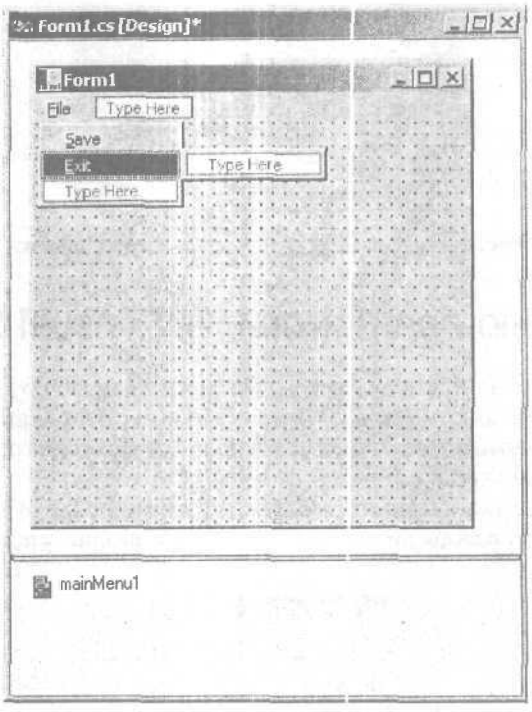


Рис. 8.26. Создание меню

Рис. 8.27. Настройка обработчика событий

При создании меню графическими средствами Visual Studio автоматически внесет необходимые изменения в служебный метод `InitializeComponent()` и добавит переменные-члены, представляющие созданные элементы меню. Если мы по-

смотрим на этот автоматически генерируемый код, я думаю, он покажется нам очень знакомым.

Что такое строка состояния

На многих формах в реальных приложениях имеются не только ниспадающие и контекстные меню, но и элемент интерфейса, называемый строкой состояния (status bar). Обычно в строке состояния выводится некоторая текстовая или графическая информация, относящаяся к работе приложения. Строка состояния может быть разделена на несколько «панелей» (pane) — отдельных частей окна. В каждой из этих панелей информация выводится отдельно.

Класс StatusBar производится напрямую от System.Windows.Forms.Control. Помимо унаследованных членов, этот класс содержит также набор свойств, которые определяют его возможности. Перечень наиболее важных свойств этого класса представлен в табл. 8.20.

Таблица 8.20. Некоторые свойства класса StatusBar

Свойство	Назначение
BackgroundImage	Позволяет получить или установить изображение, отображаемое как фон для строки состояния
Font	Позволяет получить или установить шрифт, используемый для вывода текстовой информации в строке состояния
ForeColor	Определяет цвет «переднего плана» строки состояния, то есть цвет текста
Panels	Возвращает вложенный тип StatusBarPanelCollection, который содержит все панели (объекты Panel) в строке состояния. Вся эта система очень похожа на использование коллекции для элементов меню
ShowPanels	Позволяет получить или установить значение, определяющее, будет ли та или иная панель видимой
SizingGrip	Определяет, будет ли в правом углу строки состояния показана «цеплялка» для изменения размеров строки состояния в виде треугольника с полосками

После того как объект StatusBar будет создан, следующее, что надо сделать, — поместить в него вложенные панели, то есть объекты StatusBarPanel. Для хранения этих вложенных объектов используется коллекция StatusBar$StatusBarPanelCollection. Конструктор объектов StatusBarPanel автоматически придает создаваемым панелям стандартный внешний вид (если он нас устроит, то наша задача только упростится). Некоторые наиболее важные свойства StatusBarPanel и значения для них по умолчанию представлены в табл. 8.21.

Таблица 8.21. Свойства класса StatusBarPanel

Свойство	Назначение	Значение по умолчанию
Alignment	Определяет, как будет выровнен текст на панели	HorizontalAlignment.Left
AutoSize	Определяет, будет ли панель автоматически изменять свой размер (и если будет, то как)	StatusBarPanelAutoSize.None
BorderStyle	Определяет стиль рамки вокруг строки состояния	StatusBarPanelBorderStyle.Sunken

продолжение

Таблица 8.21 (*продолжение*)

Свойство	Назначение	Значение по умолчанию
Icon	Определяет, будет ли создан значок для этой панели?	Нулевая ссылка (то есть значка нет)
MinWidth	Минимальная ширина панели	10
Style	Определяет стиль, соответствующий содержанию панели. Можно использовать только значения из перечисления StatusBarPanelStyle	StatusBarPanelStyle.Text
Text	Заголовок панели	Пустая строка
ToolTipText	Подсказка, которая будет выдана, если установить над панелью указатель мыши	То же самое
Width	Ширина строки состояния	100

Создаем строку состояния

Начнем создавать строку состояния в нашем приложении. Пусть наша строка будет состоять из двух панелей. Левая панель будет выводить текстовые сообщения, относящиеся к работе приложения, а правая — выводить системное время. Кроме того, на самый левый край левой панели мы выведем значок с рожицей (просто чтобы было интереснее). То, что мы хотим в итоге получить, представлено на рис. 8.28.

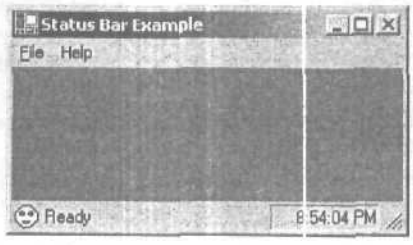


Рис. 8.28. Простая строка состояния

Для простоты мы будем продолжать работу над нашим приложением SimpleMenu, которое мы создали в предыдущем разделе. Как и любой элемент управления (тип, производный от Control), объект StatusBar должен быть добавлен в коллекцию Controls объекта Form (подробнее об этой коллекции будет сказано в главе 10). Как вы, наверное, догадываетесь, эта коллекция содержит в себе записи для всех элементов управления, размещенных на форме, в том числе и строк состояния.

Строка состояния создается следующим **образом;**

```
public class MainForm : Form
{
        // Создаем объекты для всей строки состояния и ее отдельных панелей
        private StatusBar StatusBar = new StatusBar();
        private StatusBarPanel sbPnlPrompt = new StatusBarPanel();
        private StatusBarPanel sbPnlTime = new StatusBarPanel();
```

```
public MainForm()
{
        ...
        BuildStatBar();      // Метод, который создаст нам строку состояния
                             // и настроит ее нужным образом
}

private void BuildStatBarO
{
        // Настраиваем строку состояния
        statusBar.ShowPanels = true;
        statusBar.Size - new System.Drawing.Size(212, 20);
        statusBar.Location = new System.Drawing.Point(0, 216);

        // Панели добавляются в строку состояния при помощи метода AddRange()
        statusBar.Panels.AddRange(new StatusBarPanel[] {sbPnlPrompt, sbPnlTime});

        // Настраиваем левую панель
        sbPnlPrompt.BorderStyle = StatusBarPanelBorderStyle.None;
        sbPnlPrompt.AutoSize - StatusBarPanelAutoSize.Spring;
        sbPnlPrompt.Width - 62;
        sbPnlPrompt.Text - "Ready";

        // Настраиваем правую панель
        sbPnlTime.Alignment = HorizontalAlignment.Right;
        sbPnlTime.Width = 76;

        // Добавляем значок (подробнее о них - в главе 9)
        try
        {
                // Этот значок обязательно должен находиться в каталоге приложения.
                // Как вставить ресурсы в приложение, вы узнаете в главе 9
                Icon i = new Icon("status.ico");
                sbPnlPrompt.Icon = i;
        }
        catch(Exception e)
        {
                MessageBox.Show(e.Message);
        }

        // Теперь добавляем панель управления в коллекцию Controls для формы
        this.Controls.Add(statusBar);
}
}
```

Работаем с классом Timer

Наше приложение уже работает и обе панели строки состояния на месте. Однако, как мы помним, правая панель должна показывать время. Пока же она пуста.

Чтобы правая панель нашего приложения показывала время, удобнее всего использовать класс Timer. Если у вас есть опыт работы с Visual Basic, то вы, скорее всего, уже имели дело с объектами Timer. В C++ для аналогичных целей использовалось сообщение WM_TIMER. В .NET используется объект Sys-

tem.Windows.Forms.Timer, который вызывает указанный нами (при помощи события Tick) метод через определенные интервалы (интервал задается при помощи свойства Interval). Наиболее важные члены этого класса представлены в табл. 8.22.

Таблица 8.22. Члены класса Timer

Член	Назначение
Enabled	*Это* свойство определяет, будет ли объект Timer генерировать срабатывание события Tick (можно сказать, что при помощи этого свойства объект Timer «включается» и «выключается»). Для тех же самых целей можно использовать методы Start() и Stop()
Interval	Это свойство позволяет установить промежуток между событиями Tick в миллисекундах
Start() Stop()	Эти методы используются для тех же целей, что и свойство Enabled: они управляют генерацией событий Tick
OnTick()	Этот метод может быть замещен в классах, производных от Timer
Tick	Это событие можно использовать для добавления нового обработчика событий в соответствующий MulticastDelegate

Таким образом, чтобы правая панель строки состояния начала показывать время, можно сделать так:

```
public class MainForm : Form
{
...
        private Timer timer1 = new Timer();

        public  MainForm()
        {
                // Настраиваем объект Timer
                timer1.Interval = 1000;
                timer1.Enabled = true;
                timer1.Tick += new EventHandler(timer1_Tick);
                ...
        }

        // Этот метод будет вызываться примерно каждую секунду
        private void timer1_Tick(object sender. EventArgs e)
        {
                DateTime t = DateTime.Now;
                string s = t.ToLongTimeString();

                // Выводим полученное строковое значение на правую панель
                sbPnlTime.Text = s;
        }
}
```

Обратите внимание на использование в нашей программе типа DateTime. Из этого класса можно очень просто извлечь значение системного времени (при помощи свойства Now), а затем вывести это значение (в нашем примере оно выводится на правую панель строки состояния).

Отображение в строке состояния подсказок к пунктам меню

Последнее, что мы должны сделать со строкой состояния, — настроить левую панель (которая сейчас выводит статический текст Ready) таким образом, чтобы при выборе пользователем пункта меню в строке отображалась значимая подсказка (типа This terminates the application — «Работа приложения будет завершена»).

Как мы помним, система меню в нашем примере создана заранее (в качестве примера к предыдущим разделам). Выводить подсказки для пользователей в зависимости от выбранного пункта меню проще всего, реагируя на событие Select для каждого из объектов меню. Мы организуем перехват события Select, которое будет генерироваться при выборе пользователем пунктов File ▸ Exit или Help ▸ About, и в обработчике этого события будем менять текст, выводимый в левой панели строки состояния. Помимо этого нам потребуется отслеживать момент, когда пользователь закончит свои манипуляции с меню (событие MenuComplete), чтобы вернуть текст в строке состояния в исходный вид. Вот обновление для нашей программы:

```
public class MainForm : Form
{
        ...
        public MainForm()
        {
                ...
                // Событие MenuComplete происходит при выходе пользователем из меню.
                // Мы будем реагировать на это событие, возвращая в левую панель строку
                // по умолчанию "Ready". Если мы этого не сделаем, в строке состояния
                // останется значение, которое изиенилось при выборе пользователем
                // пункта меню!
        this.MenuComplete += new EventHandler(StatusForm_MenuDone);
        BuildMenuSystem();
        }
private void FileExit_Selected(object sender, EventArgs e)
(
        sbPnlPrompt.Text = "Terminates this app";
}

private void HelpAbout_Selected(object sender. EventArgs e)
{
        sbPnlPrompt.Text = "Displays app info";
}

private void StatusForm_MenuDone(object sender. EventArgs e)
{
        sbPnlPrompt.Text = "Ready":
}

// Вспомогательные функции
private void BuildMenuSystem()
{
        // Создаем главное меню
        mainMenu = new MainMenu();
```

```
        // Создаем меню File
        MenuItem miFile = mainMenu.MenuItems.Add("&File");
        miFile.MenuItems.Add(new MenuItem("E&xit",
                             new EventHandler(this.FileExit_Clicked), Shortcut.CtrlX));

        // Обрабатываем событие Select для пункта меню Exit
        miFile.MenuItems[0].Select += new EventHandler(FileExit_Selected);

        // Теперь создаем меню Help | About
        MenuItem miHelp = mainMenu.MenuItems.Add("Help");
        miHelp.MenuItems.Add(new MenuItem("&About",
                             new EventHandler(this.HelpAbout_Clicked), Shortcut.CtrlA));

        // Обрабатываем событие Select для пункта меню About
        miHelp.MenuItems[0].Select += new EventHandler(HelpAbout_Selected);

        // Присоединяем главное меню к объекту Form
        this.Menu = mainMenu;
        }
    ...
}
```

Теперь строка состояния полностью выполняет свои функции. Как, наверно, вы уже догадываетесь, в Visual Studio предусмотрены и графические средства для создания строки состояния. После того как мы разобрались с механизмом работы строки состояния и связанными с ней объектами C#, создание строки состояния любым способом не представит для нас никакой сложности.

Код приложения StatusBar можно найти в подкаталоге Chapter 8.

Создаем панель инструментов

Последний элемент графического интерфейса формы, который мы разберем в этой главе, — это панель инструментов (ToolBar). Как правило, кнопки панели инструментов обеспечивают пользователям более легкий доступ к возможностям, которые определены в меню. Например, при нажатии пользователем на кнопку Save (Сохранить) эффект будет таким же, как и при выборе им пункта Save в меню File. Конечно же, в пространстве имен System.Windows.Forms предусмотрены типы, которые облегчают работу с панелями инструментов. Первый класс, который используется для создания панелей инструментов, — это класс ToolBar. Наиболее важные свойства этого класса представлены в табл. 8.23.

Таблица 8.23. Свойства класса ToolBar

Свойство	Назначение
BorderStyle	Определяет стиль рамки вокруг панели инструментов. Используются значения из перечисления BorderStyle
Buttons	Для работы с набором кнопок на панели инструментов (то есть с коллекцией ToolBar$ToolBarButtonCollection)
ButtonSize	Определяет размер кнопок на панели инструментов
ImageList	Возвращает элемент управления ImageList, в котором хранятся изображения, используемые на панели инструментов

та можно получить информацию о том, какая именно кнопка на панели инструментов была нажата:

```
private void ToolBar_Clicked(object sender. ToolBarButtonClickEventArgs e)
{
        // Сейчас мы просто выводим значение ToolTipText для кнопки
        MessageBox.Show(e.Button.TooltipText);
}
```

Добавление изображения на кнопки панели инструментов

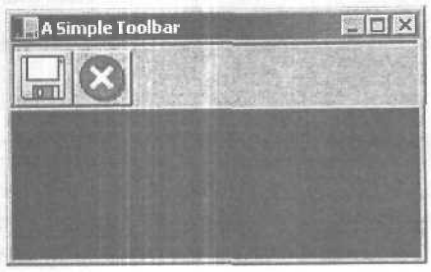
В реальных приложениях кнопки на панели инструментов гораздо чаще несут на себе не надписи, а изображения. Если мы хотим изменить наше приложение таким образом, чтобы на кнопках панели инструментов появились изображения, первое, что мы должны сделать — создать на форме объект класса ImageList. Этот класс представляет собой хранилище изображений, которое может использоваться объектами других классов (например, ToolBar). Если вы создавали панели инструментов при помощи Visual Basic 6.0, вы, без сомнения, уже знакомы с таким подходом. Давайте обновим наше приложение так, чтобы на кнопках, помимо надписей, появились еще и изображения. Выглядеть это может так:

```
public class MainForm : Form
{
        // Объект для хранения набора изображений
        private ImageList toolBarIcons = new ImageList();
        ...
        private void BuildToolBar()
        {
                // Настраиваем кнопку "Save"
                tbSaveButton.ImageIndex = 0;
                tbSaveButton.ToolTipText = "Save";

                // Настраиваем кнопку "Exit"
                tbExitButton.ImageIndex = 1;
                tbExitButton.ToolTipText = "Exit";

                // Создаем панель инструментов и добавляем на нее кнопки
                toolBar. ImageList - toolBarIcons;
                ...
                // Загружаем значки (соответствующие файлы должны быть в каталоге приложения
                toolBarIcons. ImageSize = new System.Drawing.Size(32, 32);
                toolBarIcons. Images.Add(new Icon("filesave.ico"));
                toolBarIcons. Images.Add(new Icon("fileexit.ico"));
                toolBarIcons.ColorDepth = ColorDepth.Depth16Bit;
                toolBarIcons.TransparentColor = System.Drawing.Color.Transparent;
                ...
        }
}
```

Обратите внимание на следующие моменты;

- для каждой кнопки нужно указать соответствующее ей изображение (при помощи свойства ImageIndex);
- изображения добавляются в объект ImageList при помощи метода Images.Add();
- для объекта ToolBar обязательно следует указать, какой именно объект ImageList будет использоваться при применении свойства ImageList.

Если мы запустим новый вариант нашего приложения, то увидим, что внешний вид наших кнопок значительно улучшился (рис. 8.30).



Рис, 8.30. Так панель инструментов смотрится интереснее

Если мы хотим, чтобы кнопки выглядели более стандартно, просто изменим размер изображения с 32 x 32 на 16 x 16.

Код приложения SimpleToolBar можно найти в подкаталоге Chapter 8.

Создаем панели инструментов при помощи Visual Studio IDE

В Visual Studio.NET предусмотрены средства, которые позволяют нам добавить панель инструментов при помощи графических средств. Для этого откройте Toolbox и добавьте элемент управления ToolBar на вашу форму. Добавление кнопок на панель инструментов производится при помощи окна Properties (Свойства) для панели инструментов. В строке Buttons (Кнопки) щелкните на поле Collection (Коллекция), как это показано на рис. 8.31. Откроется окно ToolBarButton Collection Editor (Редактор коллекции ToolBarButton), в котором мы сможем легко создавать кнопки и настраивать их свойства (рис. 8.32).

Рис. 8.31. Окно свойств панели инструментов

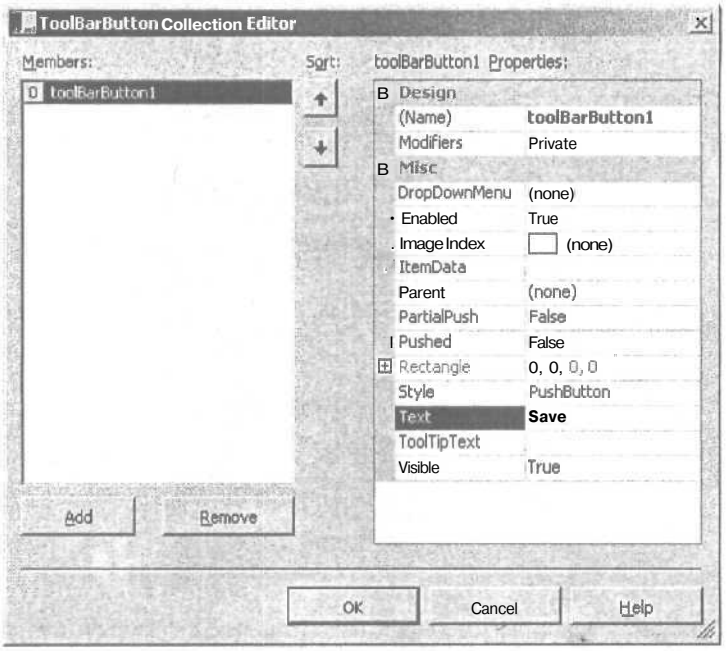


Рис. 8.32. Из этого окна производится добавление кнопок на панель инструментов

Создаем набор изображений (объект ImageList) при помощи Visual Studio

При помощи Visual Studio.NET IDE можно очень быстро не только создать панель инструментов и кнопки на ней, но и определить, какие изображения будут использоваться для кнопок. Первое, что нужно сделать — создать на форме объект ImageList. Это делается при помощи того же Toolbox (рис. 8.33).

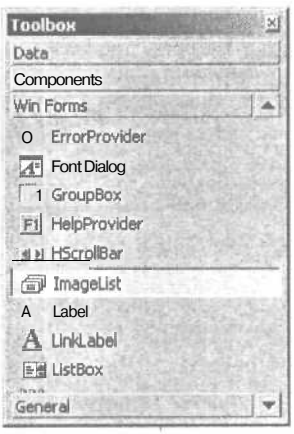


Рис. 8.33. Используем Toolbox для создания на форме объекта ImageList

Следующий шаг — открыть свойства объекта ImageList и с помощью свойства Images добавить в этот объект нужные изображения (рис. 8.34),

Рис. 8.34. Добавляем изображения в ImageList

После этого наша задача — ассоциировать панель инструментов с нужным объектом ImageList (поскольку таких объектов на форме может быть несколько). Делается это в окне свойств объекта ToolBar (рис. 8.35).

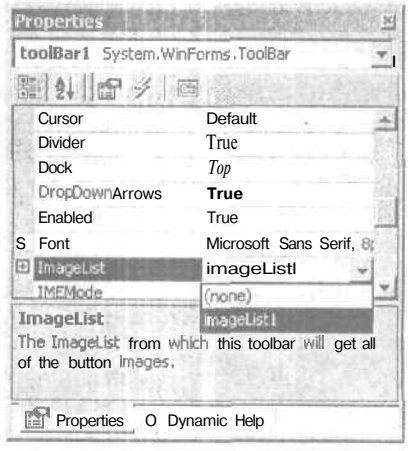


Рис. 8.35. Ассоциируем панель инструментов с объектом ImageList

И последнее, что нам осталось сделать, — вернуться в окно ToolBarButton Collection Editor (Редактор коллекции ToolBarButton) и выбрать для каждой из кнопок нужный значок (рис. 8.36).

Надеюсь, что после того как в наших примерах мы проделали все эти операции вручную, вам не составит труда разобраться с кодом, который генерирует Visual Sduio.NET IDE для панели инструментов при ее создании в графическом режиме,

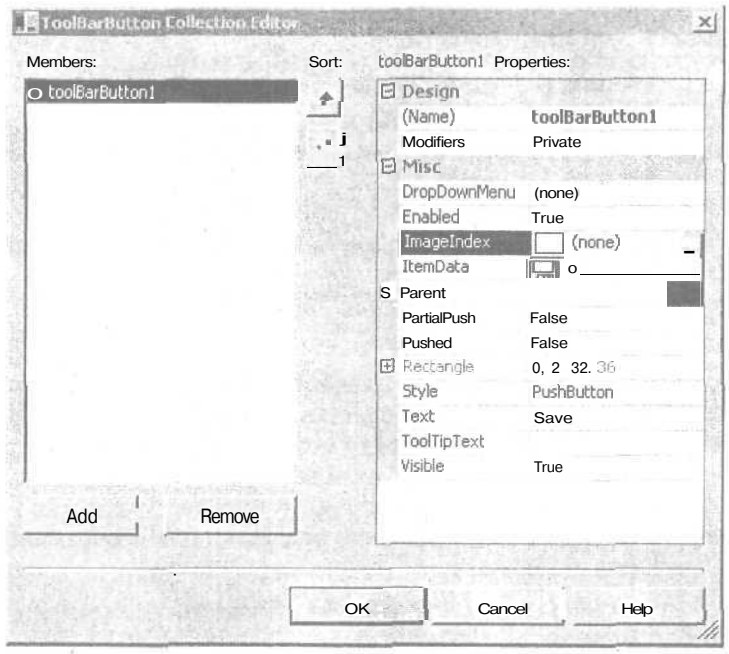


Рис. 8.36. Выбираем значки для кнопок

Пример приложения Windows Forms для работы с реестром и журналом событий Windows 2000

К этому моменту мы уже можем с легкостью создавать главную форму приложения, а на ней — главное меню, контекстные меню, строку состояния и панели инструментов. Мы используем все эти навыки в заключительном примере этой главы, Однако, чтобы было интереснее, наше приложение будет обладать дополнительными возможностями, которые могут очень пригодиться при создании реальных приложений: наша программа будет считывать и записывать данные в системны и реестр, а также взаимодействовать с журналом событий Windows 2000.

Чтобы не начинать совсем с нуля, мы продолжим работу над уже созданным нами приложением StatusBar.

Первое, что мы сделаем, — создадим новое меню верхнего уровня, которое будет называться Background Color (цвет фона). Как можно догадаться, при помощи пунктов этого меню пользователь сможет выбирать цвет фона для нашей формы. Для каждого пункта этого меню у нас будет своя подсказка, которая будет выводиться в левой панели строки состояния. Событие Clicked для всех пунктов меню Background Color будет перехватываться одним и тем же обработчиком событий ColorItem_Clicked. Точно так же событие Selected для всех пунктов меню (как мы помним, это событие нужно для отображения подсказки в строке состояния) будет перехватываться одним для всех обработчиком событий ColorItem_Selected. Вот обновление нашего код; t:

```
private void BuildMenuSystem()
{
```

```
...
        // Создаем меню Background Color
        MenuItem miColor = mainMenu.MenuItems.Add("&Background Color");
        miColor.MenuItems.Add("&DarkGoldenrod", new EventHandler(ColorItem_Clicked));
        miColor.MenuItems.Add("&GreenYellow", new EventHandler(ColorItem_Clicked));
        miColor.MenuItems.Add("&MistyRose", new EventHandler(ColorItem_Clicked));
        miColor.MenuItems.Add("&Crimson", new EventHandler(ColorItem_Clicked));
        miColor.MenuItems.Add("&LemonChiffon",new EventHandler(ColorItem_Clicked));
        miColor.MenuItems.Add("&OldLace", new EventHandler(ColorItem_Clicked));

        // Все пункты этого меню будут обрабатываться одним обработчиком события Selected
        for(int i=0; i < miColor.MenuItems.Count; i++)
            miColor.MenuItems[i].Select += new EventHandler(ColorMenuItem_Selected);
...
}
```

Когда пользователь выберет любой пункт меню Background Color, возникнет событие Select. Чтобы все-таки в строке состояния выводить ту подсказку, которая будет соответствовать выбранному пункту меню, нам необходимо предусмотреть в методе ColorMenuItem_Selected логику, отличающую пункты меню друг от друга. Мы поступим так: пусть этот метод будет извлекать текстовое имя выбранного элемента меню и передавать его в качестве подсказки для отображения в строке состояния. Выглядеть это может так:

```
private void ColorMenuItem_Selected(object sender, EventArgs e)
{
        // Определяем текстовое имя выбранного пункта меню и удаляем символ "&"
        MenuItem miClicked = (MenuItem)sender;
        string item = miClicked.Text.Remove(0,1);

        // Организуем вывод полученного значения в левую панель строки состояния
        sbPnlPrompt.Text = "Select " + item;
}
```

Помимо вывода подсказки в строку состояния нам, конечно, надо не забыть обеспечить реакцию формы на активизацию пользователем пункта меню. Как мы помним, форма должна менять цвет в зависимости от того, какой пункт меню активизировал пользователь. Это проще всего сделать при помощи свойства Form.BackColor. Имена для пунктов меню мы выбрали такие, что их (после удаления символа &) вполне можно использовать в качестве значений для свойства BackColor. Таким образом, метод ColorItem_Clicked() может выглядеть следующим образом:

```
private void ColorItem_Clicked(object sender, EventArgs e)
{
        // Определяем текстовое имя пункта меню BackColor
        MenuItem miClicked = (MenuItem)sender;

        // Удаляем символ "&"
        string color = miClicked.Text.Remove(0,1);

        // А теперь настраиваем цвет формы
        this.BackColor = Color.FromName(color);
        Color currColor = BackColor;
}
```

Нас можно поздравить: меню успешно создано и работает. Следующая наша задача — обеспечить сохранение предпочтений пользователя в системном реестре.

Взаимодействие с системным реестром

Если вы программист, работающий с COM, шансов избежать работы (обычно весьма тяжелой и трудоемкой) с системным реестром у вас нет. В мире .NET все иначе. Реестр в .NET — это не более чем удобное место хранения предпочтений пользователя. В пространстве имен Microsoft.Win32 предусмотрен набор типов, при помощи которых считать данные из реестра или внести в него какие-либо изменения можно очень легко и быстро. Краткий перечень этих типов представлен в табл. 8.25.

Таблица 8.25. Типы для взаимодействия с системным реестром

Тип Microsoft.Win32	Назначение
Registry	Можно считать этот тип высокоуровневой абстракцией всего реестра со всеми его ветвями
RegistryKey	Это — главный тип, при помощи которого производится вставка новых параметров (ключей) в реестр, их удаление и изменение
RegistryHive	Перечисление, в котором хранятся названия каждой ветви реестра

Наша цель — обеспечить сохранение предпочтений пользователя (размера шрифта и цвета фона) в системном реестре для последующего использования. Для этого мы будем использовать класс RegistryHive. Наиболее важные члены этого класса представлены в табл. 8.26.

Таблица 8.26. Члены класса RegistryKey

Член	Назначение
Name	Это свойство возвращает имя параметра реестра
SubKeyCount	Возвращает количество вложенных параметров реестра
ValueCount	Возвращает количество значений параметра реестра
Close()	Закрывает параметр реестра и записывает его в системный реестр на постоянной основе (в случае, если параметр был изменен)
CreateSubKey()	Позволяет создать новый подпараметр реестра или открыть существующий подпараметр. Принимаемое им имя подпараметра не чувствительно к реестру.
DeleteSubKey()	Удаляет указанный подпараметр реестра. Чтобы удалить подпараметр с вложенными параметрами, используйте метод DeleteSubKeyTree(). Принимаемое имя точно так же не чувствительно к регистру
DeleteSubKeyTree()	Позволяет удалить подпараметр реестра и все дерево вложенных в него параметров. Также не чувствителен к регистру
GetSubKeyNames()	Возвращает массив значений типа string, который содержит имена всех вложенных параметров
GetValue()	Многократно перегружен. Используется для получения значения параметра реестра
GetValueNames()	Возвращает массив значений типа string, представляющий имена всех значений данного параметра реестра
OpenRemoteBaseKey()	Для открытия параметра реестра на удаленном компьютере
OpenSubKey()	Перегружен. Позволяет получить параметр реестра
SetValue()	Устанавливает указанное значение. Принимаемый им параметр SubKey не чувствителен к реестру

Предположим, что у нас в меню File (Файл) появился новый пункт Save (Сохранить). При выборе пользователем этого пункта меню будет происходить следующее: вначале будет создан объект RegistryKey, а затем с его помощью текущие значения для шрифта и цвета фона будут помещены в системный реестр по адресу HKEY_CURRENT_USER\Software\Intertech\Chapter8App. Также предположим, что у нас уже объявлены две переменные: одна — для хранения информации о текущем размере шрифта (currFontSize), а вторая — для выбранного пользователем цвета (currColor). Тогда код для сохранения информации в реестре может выглядеть следующим образом (обратите внимание на применение метода RegistryKey.SetValue()):

```
// Предположим, что у нас установлены следующие значения переменных:
// Color currColor = Color.MistyRose;
// private int currFontSize = TheFontSize.Normal;

private void FileSave_Clicked(object sender. EventArgs e)
{
        // Сохраняем настройки пользователя в реестре
        RegistryKey regKey = Registry.CurrentUser;
        regKey = regKey.CreateSubKey("Software\\Intertech\\Chapter8App");
        regKey.SetValue("CurrSize", currFontSize);
        regKey.SetValue("CurrColor", currColor.Name);
}
```

Если теперь пользователь выберет цвет фона LemonChiffon, а размер шрифта — Huge (то есть 30) и после этого сохранит свои настройки, то в реестре он сможет обнаружить соответствующие параметры (рис. 8.37).

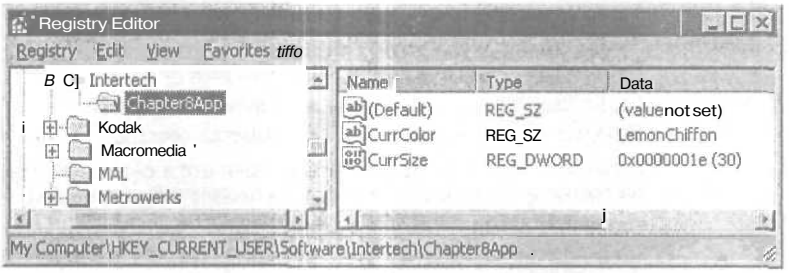


Рис. 8.37. Сохранение текущих настроек приложения в HKEY_CURRENT_USER

Для считывания информации из системного реестра используется тот же самый тип RegIstryKey. Давайте изменим конструктор нашей формы таким образом, чтобы он считывал значения, установленные в реестре, и в зависимости от этого устанавливал цвет фона и размер шрифта. Таким образом, приложение запустится с теми настройками, которые были сохранены во время предыдущего сеанса работы (обратите внимание на то, как используется метод RegistryKey.GetValue()):

```
public MainForm()
{
        // Открываем параметр реестра
        RegistryKey regKey = Registry.CurrentUser;
        regKey = regKey.CreateSubKey("Software\\Intertech\\Chapter8App");

        // Считываем значения и присваиваем их соответствующим переменным
        currFontSize = (int)regKey.GetValue("CurrSize", currFontSize);
        string c = (string)regKey.GetValue("CurrColor", currColor.Name);
```

```
        currColor = Color.FromName(c);
        BackColor = currColor;

        ...
    }
```

Наверное, у вас уже созрел вопрос: а что будет, если записей в реестре, к которым обращается наша программа, не существует? Такой вариант очень даже возможен: предположим, что пользователь запустил программу в первый раз, когда еще никаких записей в реестре просто нет. В этой ситуации, как следует из логики нашей программы, объект RegistryKey просто не сможет считать нужные ему значения из реестра!

Однако все не так страшно. Для метода GetValue() предусмотрен второй необязательный параметр (используемый в нашем примере), который будет подставляться в том случае, если запись в реестре не обнаружена. В нашем случае мы передаем текущие значения переменных currFontSize и currColor. Конечно же, в форме должна быть предусмотрена установка исходных значений этих переменных, например:

```
public class MainForm : Form
{
        Color currColor = Color.MistyRose;
        private int currFontSize = TheFontSize.Normal;
...
}
```

Последнее, что мы должны сделать, — внести изменения во вспомогательный метод BuildMenuSystem(). Цель изменений — сделать так, чтобы при выборе шрифт и против соответствующего пункта в контекстном меню устанавливался флажок. Как мы помним, сейчас у нас флажок автоматически устанавливается против пункта Normal. Поскольку ситуация изменилась, флажок должен устанавливаться в зависимости от реального положения дел. Выглядеть это может так:

```
private void BuildMenuSystem()
{
...
        // Ориентируемся на текущее значение currFontSize
        if(currFontSize = = TheFontSize.Huge)
                currentCheckedItem = checkedHuge;

        else if(currFontSize = = TheFontSize.Normal)
                currentCheckedItem = checkedNormal;

        else
                currentCheckedItem = checkedTiny;

        currentCheckedItem.Checked = true;
}
```

Взаимодействие с журналом событий Windows 2000

В Windows 2000 предусмотрена запись всех важнейших событий, происходящих в операционной системе, в журналы событий (Event Logs). Просмотреть информацию в журнале событий можно при помощи встраиваемой консоли (snap-in) Event Viewer в Microsoft Management Console (MMC). Как минимум, в системе ведется

три отдельных журнала: Application (журнал событий приложений), Security (журнал событий безопасности) и System (журнал событий системы). Журнал событий представляет важный источник информации о работе оборудования, операционной системы и установленных на компьютере приложений. Кроме того, при помощи журнала событий безопасности мы можем получать информацию для контроля — кто какие действия производил на этом компьютере, к каким ресурсам обращался и т. п. Окно Event Viewer представлено на рис. 8.38.

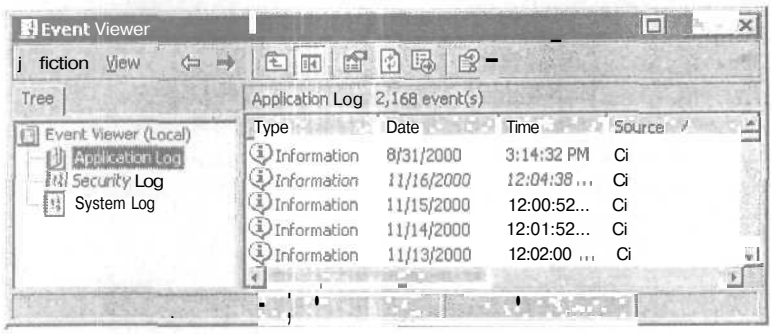


Рис. 8.38. Event Viewer в Windows 2000

Конечно же, в .NET предусмотрены типы, при помощи которых можно взаимодействовать с журналами событий программным образом. Эти типы, которые определены в пространстве имен System.Diagnostics, представлены в табл. 8.27.

Таблица 8.27. Типы System.Diagnostics, предназначенные для работы с журналами событий

Тип	Назначение
EventLog	Это — основной класс для организации взаимодействия с журналом событий Windows 2000
EventLog.EventLogEntryCollection	Коллекция для хранения типов EventLog Entry
EventLogEntry	Представляет отдельную запись в журнале событий
EventLog Names	Этот закрытый тип обеспечивает возможность указать, с каким именно журналом событий (приложений, системы или безопасности) будет работать пользователь

Члены класса EventLog позволяют записывать события во все три журнала, производить очистку журнала, считывать данные и реагировать на появление в журнале событий новых записей. Если есть необходимость, мы можем даже создать свой новый журнал событий Windows 2000 (в дополнение к трем обязательным). Наиболее важные члены класса EventLog представлены в табл. 8.28.

Таблица 8.28. Члены класса EventLog

Член	Назначение
Entries	Позволяет получить содержимое журнала событий, которое помещается в тип EventLog.EventLogEntryCollection. Эта коллекция содержит объекты класса EventLogEntry

Член	Назначение
Log	Позволяет получить или установить имя журнала событий, с которым вы работаете. Стандартно можно работать с журналами Application, System, Security, другими встроенными журналами, появляющимися после установки некоторых приложений, или пользовательским журналом событий
MachineName	Позволяет получить или установить имя компьютера, на котором работает пользователь. Если это имя не указано, по умолчанию считается, что он работает на локальным компьютере (".")
Source	Позволяет получить или установить имя приложения (source name), которое будет зарегистрировано в журнале как источник события
Clear()	Очищает все записи в журнале событий
Close()	Закрывает журнал и освобождает все ресурсы, связанные с чтением и записью информации в нем
CreateEventSource()	Устанавливает приложение как источник событий
GetEventLogs()	Создает массив журналов событий
WriteEntry()	Производит вставку записи в журнал событий

Чаще всего приложения создают записи в журнале событий приложений (Application Log) при завершении работы. Не будем отступать от этого правила и мы. Пусть в журнале событий приложений появляется новая запись при срабатывании обработчика событий FileExitClicked(). Необходимый для этого код может выглядеть следующим образом:

```
// Срабатывает при активизации пользователем пункта меню File4Ex1t
private void FileExit_Clicked(object sender, EventArgs e) ,
{
        // Мы делаем записи в журнал событий приложений ("Application"), могли бы
        // и в другой...
        EventLog log = new EventLog();
        log.Log = "Application";
        log.Source * Text;
        log.WriteEntry("Hey dude, this app shut down...");
        log.Close();

        // Завершаем работу приложения
        this.Close();
};
```

Запустим приложение и в меню File выберем Exit. Если после этого мы откроем Event Viewer, то в журнале событий приложений (Application Log) мы увидим запись, аналогичную представленной на рис. 8.39.

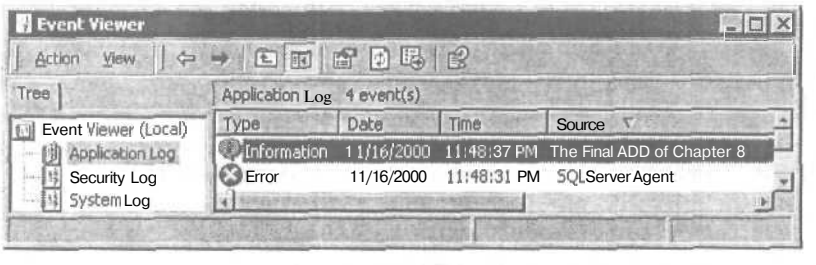


Рис. 8.39. Созданное нами событие

Чтобы просмотреть исключительно дружелюбное и многое объясняющее сообщение, которое было создано нашим приложением, просто дважды щелкнем мышью на записи (рис. 8.40).

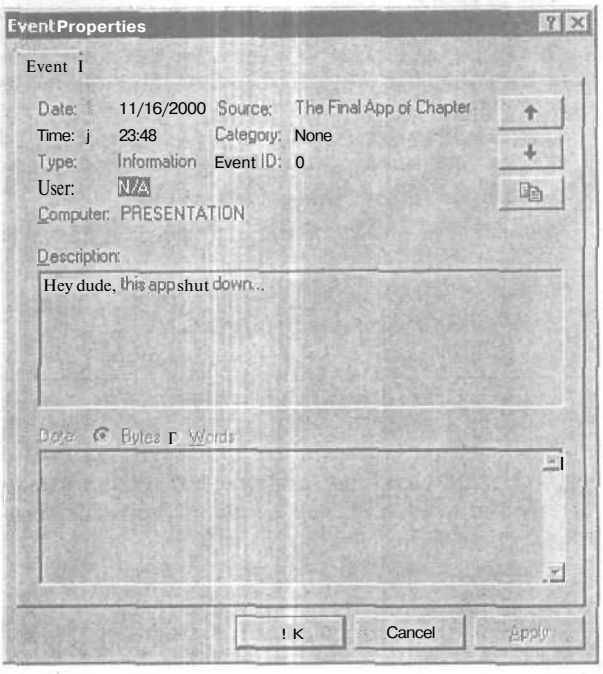


Рис. 8.40. Описание события

Как считать информацию из журнала событий

Предположим, что у нас возникла необходимость считывать информацию из журнала событий. Используя типы System.Diagnostics, мы можем сделать это так же просто, как и записать событие в журнал. В классе EventLog определено свойство, которое называется Entries. Это свойство возвращает объект класса EventLog.EventLogEntryCollection, в котором, в свою очередь, содержится набор объектов EventLogEntry (пронумерованный). Каждый объект EventLogEntry представляет собой отдельную запись в журнале событий. Наиболее важные свойства EventLogEntry представлены в табл. 8.29.

Таблица 8.29. Свойства класса EventLogEntry

Свойство	Назначение
Category	Позволяет получить текст, ассоциированный с номером CategoryNumber, для данной записи
CategoryNumber	Позволяет получить номер категории, определяемый конкретным приложением, для этой записи
Data	Позволяет получить двоичные данные, связанные с этой записью
EntryType	Позволяет получить тип записи

Свойство	Назначение
EventJD	Позволяет получить идентификатор события (также определяемый приложением) для этой записи
MachineName	Позволяет получить имя компьютера, на котором была создана данная запись
Message	Позволяет получить сообщение (на естественном языке с учетом локализации приложения), относящееся к этой записи
Source	Позволяет получить имя приложения, которое сгенерировало событие
TimeGenerated	Позволяет получить время, когдз произошло событие, приведшее к появлению записи
TineWritten	Позволяет получить время, когдз была помещена запись (локальное)
UserName	Возвращает имя пользователя, ассоциированное с событием, которое повлекло создание записи

Для простоты добавим код, считывающий данлыс из журнала событий приложения, в гот же метод FileExit_Clicked():

```
private void FileExit_Clicked(object sender. EventArgs 3)
{
    ...
    // Отображаем первые пять записей журнала событий приложений
    for(int. i = 0; i < 5; i++)
    {
        try
        f
        MessageBox.Show("Message: " + log.Entries[i].Message + "\n" + "Box пате;
                        + log.Entries[i].MachineName + "\n" + "App: " +
                        log.Entries[i].Source + "\n" + "Time entered: " +
                        log.Entries[i].TimeWritten, "Application Log
                        entry:");
        }
        catch {}
    }
}
```

Теперь при выборе пункта Exit в меню File мы увидим пять оком сообщений с информацией о записях в журнале событий приложений.

Наше готовое приложение представлено па рис. 8.41.

Код приложения FinalFormsApp можно найти в подкаталоге Chapter 8.

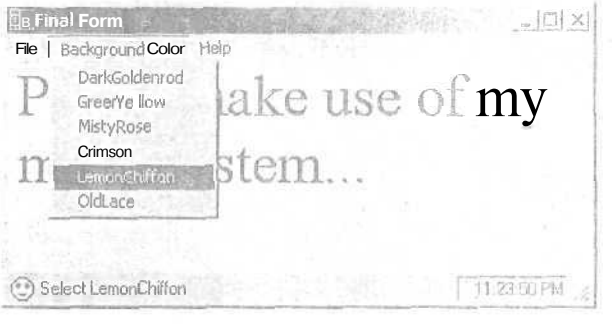


Рис. 8.41. Программа готова

Подведение итогов

В этой главе мы познакомились с искусством создания пользовательского интерфейса при помощи типов из пространства имен System.Windows.Forms. Глава начинается с основ создания главной формы приложения в C#. Мы познакомились с объектом Application и его самыми важными членами. Как мы могли убедиться, класс Form получает свои возможности путем наследования от длинной цепочки производных классов.

В этой главе мы также научились создавать ниспадающие и контекстные меню и организовывать реакцию приложения на различные события, связанные со взаимодействием пользователя с меню. Мы выяснили, как можно создавать и использовать строку состояния и панели инструментов. Кроме того, в качестве примера в главе была приведена дополнительная информация о том, как можно взаимодействовать средствами C# с системным реестром и журналом событий Windows 2000.

Графика становится лучше (GDI+) 9

В предыдущей главе мы научились создавать главное окно приложения с помощью типов из пространства имен System.Windows.Forms. Теперь наша задача — научиться работать с выводом на форму графики (изображения и текста).

Мы начнем с обзора многочисленных пространств имен, предназначенных для работы с графикой в .NET. Мы узнаем, как инициировать сеансы вывода графики и как реагировать на эти сеансы в приложении. Кроме того, мы рассмотрим многочисленные возможности объекта Graphics и приемы работы с ним. После того как общее представление о процессе вывода графики на форму будет получено, мы займемся непосредственно процессом управления цветом, шрифтами, вывода геометрических фигур и растровых изображений. При этом нам не обойтись без знакомства с такими типами, как Brush, Pen, Color, Point и Rectangle (а также множеством других). Мы рассмотрим и такие возможности GDI+, как реакция на «попадание» в непрямоугольную область и перетаскивание (drag-and-drop).

Глава завершается анализом нового формата хранения ресурсов приложений в .NET. Мы узнаем, как помещать в сборку внешние ресурсы, познакомимся с пространством имен System.Resources и выполнением операций чтения и записи для файлов *.resx, а также с тем, как можно извлекать ресурсы из сборки во время выполнения при помощи типа ResourceManager.

Обзор пространств имен GDI+

GDI расшифровывается как Graphic Device Interface (интерфейс графических устройств). Этим словом обозначается подсистема Windows, предназначенная для вывода графических изображений (а Windows вся основана на использовании графики) на экран и на принтер. GDI+ — это новый набор программных интерфейсов, используемый в .NET.

В .NET предусмотрено множество пространств имен, предназначенных для вывода двумерных графических изображений. Помимо ожидаемых стандартных

типов (например, для работы с цветом, со шрифтами, с пером и кистью, с изображениями) в этих пространствах имен предусмотрены типы для выполнения достаточно изощренных операций, таких как геометрические преобразования, сглаживание неровностей, подготовка палитры, поддержка вывода на принтер и многие другие. Перечень наиболее важных пространств имен для работы с графическими изображениями представлен в табл. 9.1.

Таблица 9.1. Наиболее важные пространства имен GDI+

Пространство имен	Специализация
System.Drawing	Это — важнейшее пространство имен GDI+, которое содержит основный типы для вывода графики (для работы со шрифтами, перьями, кистью и т. п.), а также исключительно важный тип Graphics
System.Drawing.Drawing2D	В этом пространстве имен предусмотрены типы для выполнения более сложных операций с двумерной графикой (градиентная заливка, геометрические преобразования и т. п.)
System.Drawing.Imaging	Здесь определены типы, которые позволяют напрямую работать с графическими изображениями (менять палитру, извлекать метаданные изображений, выполнять операции с метафайлами и т. п.)
System.Drawing.Printing ,	Это пространство имен определяет типы для вывода графики на принтер и взаимодействия с принтером в целом
System.Drawing.Text	Это пространство имен позволяет работать с системными шрифтами. Например, как будет показано на примерах этой главы, тип FontCollection позволяет получать список всех установленных в системе шрифтов

Чтобы обеспечить возможность работы с графикой в нашем приложении, необходимо добавить в него ссылку на сборку System.Drawing.dll. Если мы воспользуемся шаблоном Windows Application, то эта ссылка будет добавлена автоматически. После этого нам необходимо добавить в список используемых пространств имен строку

```
using System.Drawing;
```

и можно приступать к работе с графикой.

Пространство имен System.Drawing

Подавляющее большинство типов для работы с графикой, которые нам потребуются, находятся именно в пространстве имен System.Drawing. Некоторые наиболее важные типы этого пространства имен представлены в табл. 9.2.

Для значительной части этих типов используются перечисления, которые также определены в пространстве имен System.Drawing. Некоторые из этих перечислений представлены в табл. 9.3.

Если у вас есть опыт работы с библиотеками типов для работы с графикой в других языках (например, в Java), возможности System.Drawing покажутся вам очень знакомыми. Следующая наша задача - - познакомиться с основными служебными типами, которые используются в GDI+.

Таблица 9.2, Типы пространства имен System.Drawing

Тип	Назначение
Bitmap	Инкапсулирует файл изображения и определяет набор методов для выполнения различных операций с этим изображением
Brush Brushes SolidBrush SystemBrushes TextureBrush	Объекты Brush (кисть) используются для заполнения пространства внутри геометрических фигур (например, прямоугольников, эллипсов или многоугольников). Тип Brush — это абстрактный базовый класс, остальные типы являются производными от Brush и определяют разные наборы возможностей. Дополнительные типы Brush определены в пространстве имен System.Drawing.Drawing2D
Color SystemColors ColorTranslator	Структура Color определяет набор статических полей, которые могут быть использованы для настройки цвета. Тип ColorTranslator позволяет создавать новый цвет (объект Color) на основе внешнего представления этого цвета (цвета Win32, типа OLE_COLOR, констант цвета в HTML и т. п.)
Font FontFamily	Тип Font инкапсулирует характеристики шрифта (имя, размер, начертание и т. п.). FontFamily представляет набор шрифтов, которые относятся к одному семейству, но имеют некоторые небольшие отличия
Graphics	Этот важнейший класс определяет набор методов для вывода текста, изображений и геометрических фигур. Можно считать этот тип эквивалентом типа HDC в Win32
Icon SystemIcons	Эти классы предназначены для работы с пользовательскими и системными значками
Image ImageAnimator	Image — это абстрактный базовый класс, который обеспечивает возможности типов Bitmap, Icon и Cursor. ImageAnimator позволяет производить показ изображений (типов, производных от Image) через указанные вами интервалы времени
Pen Pens SystemPens	Pen (перо) — это класс, при помощи которого можно рисовать прямые и кривые линии. В классе Pen определен набор статических свойств, при помощи которых можно получить объект Pen с заданными свойствами (например, с установленным цветом)
Point PointF	Эти структуры обеспечивают работу с координатами точки. Point работает со значениями типа int, а PointF — со значениями типа float
Rectangle RectangleF	Эти структуры предназначены для работы с прямоугольными областями (int/float)
Size SizeF	Эти структуры обеспечивают работу с размерами (высотой и шириной). Size использует значения типа int, a SizeF — типа float
StringFormat	Этот тип используется для форматирования текста (определения выравнивания, междустрочного интервала и т. п.)
Region	Определяет область, занятую геометрической фигурой

Таблица 9.3. Перечисления пространства имен System.Drawing

Перечисление	Назначение
ContentAlignment	Определяет расположение содержимого в области вывода (слева, справа, по центру и т. п.)
FontStyle	Определяет свойства шрифта (полужирный, курсив и т. п.)
GraphicsUnit	Определяет единицы измерения для графического элемента (аналогично константам режима отображения в Win32)
KnownColor	Определяет дружественные имена системных цветов

продолжение

Таблица 9.3 ***(продолжение)***

Перечисление	Назначение
StringAlignment	**Определяет выравнивание текстовой строки**
StringFormatFlags	**Определяет** форматирование **текстовых** строк **(например, содержит** значения NoWrap, LineLimit **и т. п.)**
StringTrimminig	**Определяет, как будут обрезаться строки, которые не помещаются полностью в отведенной им области**
StringUnit	**Определяет** единицы **измерения для строки текста**

Служебные типы System.Drawing

Многие методы, определенные в классе Graphic, требуют, чтобы мы указали положение или область для вывода графического объекта. Например, при использовании метода DrawString() необходимо указать местонахождение выводимой текстовой строки на элементе управления. Метод DrawString() многократно перегружен, но один из наиболее часто используемых вариантов требует указания координат (x, y) или прямоугольной области для вывода. Другим часто используемым методам необходимо передавать размеры (высоту и ширину) прямоугольной области, в которую будет производиться вывод, или, если область вывода будет не прямоугольной, задать эту область другим способом.

Для передачи методам подобной информации в пространстве имен System.Drawing предусмотрены типы Point, Rectangle, Region и Size. Point используется для передачи координат (x, y). Rectangle определяет координаты двух точек, которые будут восприняты как верхний левый и нижний правый углы прямоугольника. Тип Size определяет размер прямоугольной области в каком-либо измерении (то есть используется для указания высоты или ширины). Тип Region необходим для работы с непрямоугольными областями.

Внутренние переменные, используемые для хранения данных в Point, Rectangle и Size, являются целочисленными (то есть относятся к типу int). Если же у нас возникла необходимость указывать координаты или размеры при помощи значений с плавающей запятой (тип float), то в нашем распоряжении типы PointF, RectangleF и SizeF. Они, как несложно догадаться, отличаются лишь форматом внутренних переменных (float вместо int). Наборы членов у них полностью одинаковы.

Тип Point(F)

Первый служебный тип, который мы рассмотрим, - тип System.Drawing.Point(F). Как вы помните, мы уже создавали наш собственный класс Point в качестве примера в главе 5. Можно считать наш вариант этого класса упрощенным вариантом Point, определенного в System, Drawing. Наиболее важные члены этого класса представлены в табл. 9.4.

Несмотря на то что этот тип чаще всего используется в приложениях **GDI+** с графическим пользовательским интерфейсом, его вполне можно использовать в приложениях любого типа. Например, ниже приведено консольное приложение, которое использует тип System.Drawing.Point:

```
namespace DrawingUtilTypes
{
using System:
```

```
using System.Drawing; // Это пространство имен нужно для работы с типами GDI+

public class UtilTypes
{
        public static int Main(string[] args)
        {
                // Создаем точку, а затем смещаем ее
                Point pt = new Point(100, 72);

                System.Console.WriteLine(pt);
                pt.Offset(20, 20);
                System.Console.WriteLine(pt);

                // Применяем перегруженные операторы Point
                Point pt2 = pt;

                if (pt = = pt2)
                        Console.WriteLine("Points are the same");
                else
                        Console.WriteLine("Different points");

                // Меняем значение координаты X для pt2
                pt2.X = 4000;

                // А теперь выводим информацию о координатах каждой точки:
                Console.WriteLine("First point: {0}", pt.ToString());
                Console.WriteLine("Second point: {0}", pt2.ToString());
                return 0;
        }
}
```

Таблица 9.4. Члены типа Point (PointF)

Член	Назначение
+ – == !=	Перегруженные операторы для выполнения различных действий с координатами x и y
x y	Эти свойства позволяют получать и устанавливать значения координат x и y
IsEmpty	*Это* свойство возвращает true, если значения x и y равны нулю
Offset()	Этот метод позволяет произвести смещение точки относительно исходной позиции

Результат работы этой программы представлен на рис. 9.1.

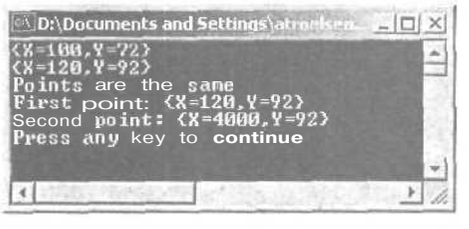


Рис. 9.1. Применение класса из System.Drawing в консольном приложении

Тип Rectangle(F)

Как и тип Point, Rectangle может пригодиться нам в любом приложении, но чаще всего этот класс используется в приложениях GDI+. Наиболее важные члены этого класса представлены в табл. 9.5.

Таблица 9.5. Члены классов Rectangle и RectangleF

Член	Назначение
= = !=	Перегруженные операторы, позволяющие сравнивать два прямоугольника
Inflate() Intersect() Union()	Эти статические методы позволяют увеличивать размеры прямоугольника и создавать новые прямоугольники путем разделения или объединения существующих
Top Left Bottom Right	Эти свойства устанавливают измерения прямоугольника
Height Width	Эти свойства определяют высоту и ширину прямоугольника
Contains()	Этот метод позволяет определить, попадает ли точка с указанными координатами (или другой прямоугольник) внутрь области, занимаемой прямоугольником. Используется очень часто
X Y	Определяют координаты *x* и y верхнего левого угла прямоугольника

В классе Rectangle() предусмотрен исключительно удобный метод Contains(). Этот метод позволяет определить, попадает ли точка или другой прямоугольник с указанными вами координатами в область, занятую прямоугольником, с которым мы работаем. В этой главе мы рассмотрим использование этого метода для определения «попаданий» внутрь изображений GDI+. А пока — простой пример применения этого метода:

```
public static int Main(string[] args)
{
        ...
        Rectangle r1 = new Rectangle(0, 0, 100, 100);
        Point pt3 - new Point(101, 101);

        if (r1.Contains(pt3))
              Console.WriteLine("Point is within the rect!");
        else
              Console.WriteLine("Point is not within the rect!");

        // Теперь помещаем точку внутрь области прямоугольника
        pt3.X = 50;
        pt3.Y = 30;

        if(r1.Contains(pt3))
              Console.WriteLine("Point is within the rect!");
        else
              Console.WriteLine("Point is not within the rect!");

        return 0;
}
```

Тип Size(F)

Типы Size и SizeF очень просты в применении и практически не требуют комментариев. Эти структуры обеспечивают работу с размерами. Краткий перечень членов этих типов представлен в табл. 9.6.

Таблица 9.6. Члены типов Size и SizeF

Член	Назначение
+ - == !=	Перегруженные операторы
Height Width	Эти свойства определяют высоту и ширину — два размера, с которыми работает Size(F)

Класс Region

Последний из служебных типов System.Drawing, который мы рассмотрим, — это класс Region. Этот класс представляет собой внутреннюю область, занятую геометрической фигурой. Конечно, чтобы создать объект этого класса, нам придется передать его конструктору некоторый объект, представляющий собой геометрическую фигуру. Например, предположим, что у нас есть прямоугольник размером 100 на 100 пикселов. Чтобы получить объект класса Region, соответствующий внутренней области этого прямоугольника, код может быть таким:

```
// Получаем объект Region для прямоугольника
Rectangle r = new Rectangle(0. 0. 100. 100);
Region rgn = new Region(r);
```

После того как объект класса Region создан, мы можем использовать многочисленные члены этого класса. Наиболее важные из них представлены в табл. 9.7.

Таблица 9.7. Члены класса Region

Член	Назначение
Complement()	Дополняет объект Region другими графическими объектами, которые не пересекаются с исходным объектом Region
Exclude()	Исключает область, занимаемую другим графическим объектом, из области объекта Region
GetBounds()	Возвращает объект класса RectangleF, представляющий прямоугольник, в который точно вписана область, занимаемая объектом Region
Intersect()	Перегружен. Уменьшает область, занимаемую исходным объектом Region, до области наложения друг на друга исходного и указанного пользователем объектов Region
IsEmpty() MakeEmpty()	Позволяют определить, имеет ли область, занимаемая данным объектом Region, нулевой размер, или установить нулевой размер для области Region
IsInfinite() MakeInfinite()	Позволяют определить, является ли область, занимаемая объектом Region, бесконечной, или установить бесконечный размер для данной области
Transform()	Преобразует область объекта Region на основе передаваемого объекта Matrix

продолжение

Таблица 9.7 (*продолжение*)

Член	Назначение
Translate()	Сдвигает координаты объекта Region на указанную пользователем величину
Union()	Объединяет указанный объект Region с другим графическим объектом
Xor()	Объединяет указанный объект Region с другим графическим объектом, исключая при этом область пересечения этих двух объектов

Мы получили общее представление о служебных типах пространства имен System.Drawing. На протяжении этой главы мы еще не раз встретимся с этими типами. А сейчас — вперед, к сеансам вывода графики в .NET!

Код приложения UtilTypes можно найти в подкаталоге Chapter 9.

Сеансы вывода графики

В предыдущей главе мы говорили о том, что в классе Control определен виртуальный метод OnPaint(). Если мы хотим, чтобы на главной форме нашего приложения (или элементе управления, или любом другом классе, производном от Control) выводилась графика, нам потребуется заместить этот метод и извлечь объект Graphics из передаваемых этому методу параметров:

```
public class MainForm : Form
{
        public MainForm()
        {
                CenterToScreen();
                this.Text = "Basic Paint Form";
        }

        public static void Main(string[] args)
        {
                Application.Run(new MainFormO);
        }

        protected override void OnPaint(PaintEventArgs e)
        {
                Graphics g = e.Graphics;

                g.DrawString("HelloGDI+", new Font("Times New Roman", 20),
                                          new SolidBrush(Color.Black), 0. 0);
        }
}
```

Для вывода изображения на форму в нашем распоряжении есть два способа. Первый (который приведен выше) заключается в том, что мы напрямую дописываем необходимую логику в метод OnPaint(). Второй (более предпочтительный) предусматривает перехват события Paint. Для реализации второго способа предыдущее определение класса можно изменить следующим образом:

```
public class MainForm : Form
{
        public MainFormO
        {
                ...
```

```
            // Добавляем обработчик события
            this.Paint += new System.Windows.Forms.PaintEventHandler(MainForm_Paint);
        }

        // Обратите внимание на сигнатуру обработчика событий
        public void MainForm_Paint(object sender. PaintEventArgs e)
        {
            Graphics g · e.Graphics;
            ...
        }

        public static void Main(string[] args)
        {
            Application.Run(new MainForm());
        }
}
```

Вне зависимости от того, какой метод мы выберем, механизм вывода графики будет одним и тем же. Как только наше приложение станет «грязным», в очереди сообщений для этого приложения появится специальное сообщение, инициирующее сеанс вывода графики (перерисовки). Приложение становится «грязным» тогда, когда изменяются его размеры, когда его окно полностью или частично перекрывается окном другого приложения, когда оно восстанавливается из минимизированного состояния. В результате либо вызывается метод `OnPaint()`, либо срабатывает наш обработчик события `Paint`.

Как сделать клиентскую область вашего приложения «недействительной»

Иногда возникает необходимость инициировать перерисовку изображения на форме вручную (иными словами, программным образом поместить в очередь сообщений сообщение о необходимости перерисовки). Например, в нашей программе может быть предусмотрено диалоговое окно для выбора пользователем изображения. После того как пользователь выберет то, что ему нужно, форму необходимо будет перерисовать, чтобы отобразить выбранное изображение.

Для того чтобы инициировать перерисовку формы программным образом, используется метод `Invalidate()`. Вызов этого метода может выглядеть так;

```
public class MainForm ; Form
{
...
    private void MainForm_Paint(object sender. PaintEventArgs e)
    {
        Graphics g - g.Graphics;
        // Логика для вывода изображения
        ...
    }

    private void GetNewBitmap()
    {
        // Выводим окно диалога и получаем выбранное пользователем изображение
        ...
        // А теперь перерисовываем клиентскую часть формы
        Invalidate();
    }
}
```

Существует множество перегруженных вариантов метода Invalidate(). Например, мы можем указать конкретную прямоугольную область, подлежащую перерисовке, вместо всей клиентской части формы (она перерисовывается по умолчанию). Если нам потребуется в приложении перерисовать лишь верхнюю левую часть формы, выглядеть это может так:

```
// Перерисовать указанную нами прямоугольную область на форме
private void UpdateUpperArea()
{
        Rectangle myRect = new Rectangle(0, 0. 75, 150):
        Invalidate(myRect);
]
```

Выводим графические объекты без события Paint

Очень часто бывает так, что вывод графического объекта необходимо произвести не в стандартных ситуациях, когда возникает событие Paint, а в ответ на другие события. Например, предположим, что наша задача — вывести маленький кружок в том месте, где на форме был сделан щелчок мышью. Первая наша задача, как всегда при выводе изображений, — получить объект Graphics, а затем выполнить с этим объектом необходимые манипуляции. Объект Graphics можно получить при помощи метода Graphics.FromHwnd(). Обратите внимание, что единственный параметр, передаваемый этому методу, — это значение свойства Handle. Свойство Handle, как говорилось в предыдущей главе, определено в классе Control и наследуется всеми классами, производными от Control. Работа с объектом Graphics может выглядеть в нашем примере так:

```
private void MainForm_MouseDown(object sender, MouseEventArgs e)
{
        // Получаем объект Graphics
        Graphics g = Graphics.FromHwnd(this.Handle);

        // Теперь в месте щелчка мышью рисуем кружок диаметром 10 пикселов
        g.DrawEllipse(new Pen(Color.Green), e.X. e.Y. 10. 10);
}
```

Легко убедиться на практике, что при любой перерисовке формы (возникновении события Paint) все кружки, выведенные на форме после щелчков мышью, исчезнут. Это не удивительно, поскольку, как мы видим, вывод графических объектов (кружков) в нашем случае происходит только в результате щелчка мышью, но не при выводе формы на экран.

Скорее всего, в реальном приложении потребуется не только выводить какие-либо графические объекты на форму, но и сохранять их при перерисовке формы — мы же не хотим, чтобы они пропадали, например, при перекрытии формой окном другого приложения? Поэтому нам придется позаботиться о том, чтобы информация о графических объектах каким-то образом сохранялась и использовалась при перерисовке формы. Самый простой способ — создать внутреннюю коллекцию (например, при помощи класса ArrayList) и помещать туда нужные нам объекты. Затем к этой коллекции будет обращаться метод OnPaint(). В нашей ситуации проще всего использовать коллекцию объектов Point, поскольку для рисования кружков ничего, кроме координат их центра, нам не нужно:

```
public class MainForm : System.Windows.Forms.Form
{
        // Коллекция для хранения координат всех кружков
        private ArrayList myPts = new ArrayList();
        ...

        private void MainForm_MouseDown(object sender. MouseEventArgs e)
        {
                // Получаем объект Graphics
                Graphics g = Graphics.FromHwnd(this.Handle);

                // Раньше мы рисовали кружки по щелчку мыши:
                //      g.DrawEllipse(new Pen(Color.Green). e.X. e.Y. 10, 10):

                // А теперь мы просто добавляем координаты указателя при щелчке
                // в коллекцию и вызываем Invalidate()
                myPts.Add(new Point(e.X, e.Y)):
                InvalidateO;
        }

        private void MainForm_paint(object sender, PaintEventargs e)
        {
                Graphics g = e.Graphics;
                g.DrawString("Hello CDI+". new Font("Times New Roman". 20).
                                        new SolidBrush(Color.Black). 0. 0);

                // Выводим все точки
                foreach(Point p in myPts)
                        g.DrawEllipse(new Pen(Color.Green). p.X. p.Y. 10. 10):
        }
}
```

Таким образом, использование метода Graphics.FromHwnd() — это очень удобный способ получить объект Graphics без использования события Paint. Результат работы нашей программы представлен на рис. 9.2. Щелкайте себе на здоровье!

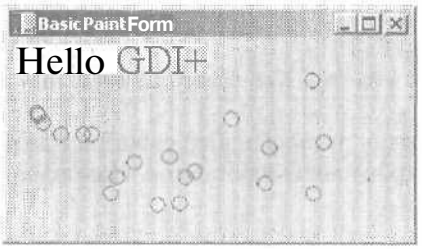


Рис. 9.2. Простое приложение GDI+

Код приложения BasicPaintForm можно найти в подкаталоге Chapter 9.

Возможности класса Graphics

Мы уже знаем, как можно получить объект Graphics в различных ситуациях. Следующее, что нам нужно сделать, — научиться в полной мере использовать его возможности. Все возможности вывода изображений в GDI+ сосредоточены именно

в этом классе. Можно считать этот класс неким виртуальным устройством, на которое производится вывод графики. Для тех, кто имеет опыт работы с традиционным программированием под Windows, еще раз напомним, что это класс можно считать аналогом HDC (Handle to Device Context, логический номер контекста устройства) в Win32. В Graphics определено очень большое количество методов для вывода текста, изображений, геометрических фигур и т. п. Наиболее часто используемые методы этого класса представлены в табл. 9.8.

Таблица 9.8. Некоторые методы класса Graphics

Метод	Назначение
FromHdc() FromHwnd() FromImage()	Эти статические методы обеспечивают возможность получения объекта Graphics из элемента управления или изображения
Clear()	Заполняет объект Graphics выбранным пользователем цветом, удаляя его предыдущее содержимое
DrawArc() DrawBezier() DrawBeziers() DrawCurve() DrawEllipse() DrawIcon() DrawLine() DrawLines() DrawPie() DrawPath() DrawRectange() DrawRectangles() DrawString()	Эти методы (как и многие другие) предназначены для вывода изображений и геометрических фигур
FillEllipse() FillPath() FillPie() FillPolygon() FillRectangle()	Эти методы предназначены для заполнения внутренних областей графических объектов
MesureString()	Возвращает структуру Size, представляющую границы блока текста

В классе Graphics также определен набор свойств, при помощи которых можно настраивать параметры графических объектов. Наиболее важные свойства представлены в табл. 9.9.

Таблица 9.9. Свойства класса Graphics

Свойство	Назначение
Clip Graphics ClipBounds VisibleClipBounds IsClipEmpty IsVisibleClipEmpty	Эти свойства позволяют настроить параметры отсечения для объекта
Transform	Для проведения преобразований координат (более подробно — см. далее в этой главе)

Свойство	Назначение
PageUnit PageScale DpiX DpiY	Эти свойства позволяют задать «место происхождения» операции вывода графического объекта и единиц измерения
SmoothingMode PixelOffsetMode TextRenderingHint	Позволяют задать плавность переходов для геометрических объектов и текста. Устанавливаются при помощи значений из соответствующих перечислений, определенных в пространствах имен System.Drawing и System.Drawing2D
Compositing Mode CompositingQuality	Свойство Compositing Mode определяет, будет ли выводимый графический объект выводиться над фоном или будет происходить смешение с фоном. Используются значения из перечисления CompositingMode, определенном в пространстве имен System.Drawing2D. CompositingQuality определяет параметры процесса смешивания. Для него используются значения из перечисления CompositingQuality, также определенном в пространстве имен System.Drawing2D
InterpolationMode	Определяет интерполяцию между конечными точками. Используются значения из соответствующего перечисления

Мы будем использовать многие из этих свойств в примерах этой главы.

Система координат по умолчанию в GDI+

Перед тем как мы начнем знакомиться с тонкостями вывода графических объектов, нам необходимо разобраться в используемой в GDI+ системе координат. Как и в API Win32, в GDI+ мы можем выбирать из множества систем координат. Система, принятая по умолчанию, использует в качестве единицы измерения пикселы, а в качестве исходной точки — верхний левый угол. Координата X определяет смещение вправо, а координата Y — смещение вниз (рис. 9.3).

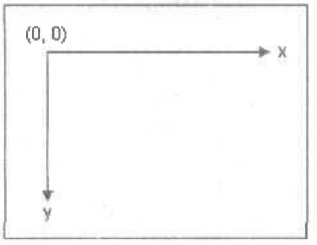


Рис. 9.3. Система координат, используемая по умолчанию

Например, если мы произведем вывод прямоугольника следующим образом:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
        // Выводим прямоугольник, используя систему координат по умолчанию
    e.Graphics.DrawRectangle(new Pen(Color.Red, 5), 10, 10, 100, 100);
}
```

то получится прямоугольник размером 90 x 90 пикселов, отстоящий от верхнего левого края формы на 10 пикселов вниз и вправо (рис. 9.4).

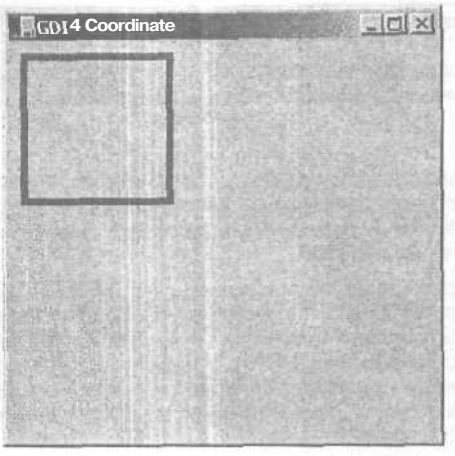


Рис. 9.4. Вывод изображения с применением системы координат по умолчанию

Скорее всего, в реальном приложении мы будем использовать именно систему координат по умолчанию. Однако нам ничто не мешает в случае необходимости воспользоваться и альтернативной системой.

Применение альтернативных единиц измерения

Как уже говорилось, единица измерения по умолчанию — это пиксел. Однако при помощи свойства PageUnit объекта Graphics мы можем выбрать другую единицу измерения, которая будет применяться к этому объекту. Для свойства PageUnit используются значения из перечисления GraphicsUnit (табл. 9.10).

Таблица 9.10. Перечисление GraphicsUnit

Значение	Используемая единица измерения
Display	1/75 часть дюйма
Document	1/300 часть дюйма
Inch	Дюйм
Millimeter	Миллиметр
Pixel	Пиксел
Point	1/72 часть дюйма

Если мы изменим код нашего примера следующим образом:

```
private void MainForm_Paint(object sender. PaintEventArgs e)
{
        // Выводим прямоугольник... теперь в дюймах, а не пикселах
        e.Graphics.PageUnit = GraphicsUnit.Inch;
        e.Graphics.DrawRectangle(new Pen(Color.Red, 5). 0. 0, 100, 100);
}
```

то в результате он станет просто неузнаваем (рис. 9.5).

Если вам трудно понять, что же произошло, объясняем: примерно 85 % клиентской площади формы теперь занято рамкой от прямоугольника. Мы указали тол-

щину пера (Реп) равной 5. Теперь это значит, что наш 100-дюймовый прямоугольник будет нарисован пером 5-дюймовой толщины. Фактически на нашей форме поместилась лишь крохотная часть внутренней области прямоугольника. Это — серая часть в правом нижнем углу.

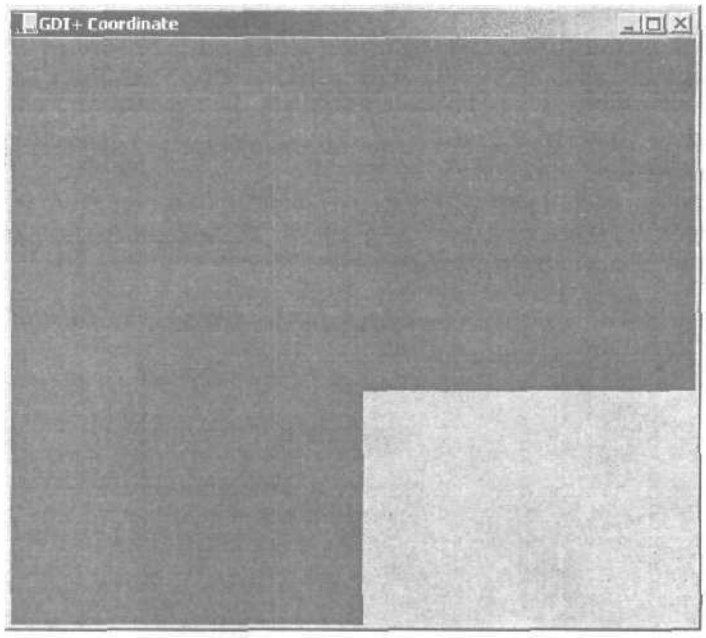


Рис. 9.5. То же самое, но единицы измерения — дюймы

Применение альтернативных точек отсчета

Как мы помним, по умолчанию точкой отсчета для системы координат является верхний левый угол клиентской площади формы. В подавляющем большинстве случаев это нас вполне устроит. Однако бывает ситуации, когда удобнее, чтобы точка отсчета системы координат была расположена в другом месте. Например, предположим, что в нашем приложении необходимо резервировать 100-пиксельную зону по всему периметру клиентской области формы.

Конечно, в этой ситуации мы можем также воспользоваться системой отсчета по умолчанию, а смещение при выводе всех графических объектов обеспечивать вручную. Но гораздо проще и надежнее сместить систему координат таким образом, чтобы вывод графических объектов начинался с отметки (100, 100) всистеме координат по умолчанию. По крайней мере, при использовании этого подхода нам придется обеспечить смещение точки отсчета координат единственный раз, а всю остальную логику, отвечающую за вывод графических объектов, можно будет оставить в неприкосновенности.

В GDI+ для изменения точки отсчета системы координат используется метод TranslateTransform(), определенный в классе Graphics. Например, установить точку отсчета в положение 100, 100 относительно системы координат по умолчанию можно следующим образом:

```
private void MainForm_Paint(object sender. PaintEventsArgs e)
{
        // Настраиваем единицы измерения
        e.Graphics.PageUnit = GraphicsUnit.Point;

        // Настраиваем новую точку отсчета для системы координат
        e.Graphics.TranslateTransform(100. 100);

        // Код для вывода прямоугольника остается прежним
        e.Graphics.DrawRectangle(new Pen(Color.Red. 1), 0. 0, 100. 100):
}
```

В качестве примера можно воспользоваться приложением CoorSystem, которое находится в подкаталоге Chapter 9. В этом приложении созданы два меню верхнего уровня, при помощи которого мы можем выбрать используемые при выводе прямоугольника единицы измерения и точку отсчета для системы координат. Окно этого приложения представлено на рис. 9.6.

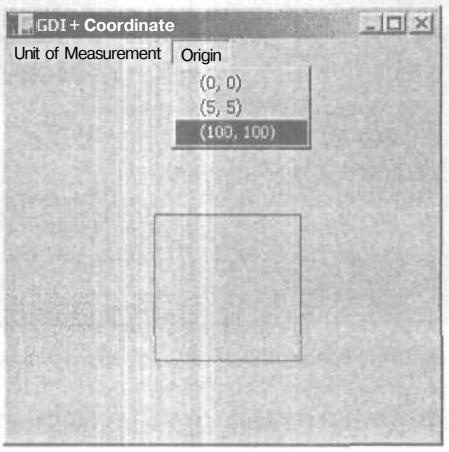


Рис. 9.6. Тестовое приложение для выбора единиц измерения и точки отсчета координат

В этой главе мы познакомимся с многими другими возможностями объекта Graphics. А теперь наша тема — работа с цветом в GDI+.

Работа с цветом

При использовании многих методов вывода, которые определены в классе Graphi cs, мы должны указать используемый цвет. Как правило, для этого используется структура Color. Эта структура позволяет задать цвет в системе ARGB (aplha-red-green-blue, альфа-канал (отвечающий за прозрачность) — красный — зеленый — синий). Чаще всего для выбора цвета используются статические свойства этой структуры, которые возвращают объект типа Color:

```
// Один из множества предопределенных цветов
Color c = Color.PapayaWhip;
```

Как видно из табл. 9.11, существуют и другие способы, при помощи которых мы можем создать объект Color. Вне зависимости от того, какой из них мы исполь-

зуем, члены структуры Color позволят нам получить полную информацию о выбранном нами цвете (табл. 9.11).

Таблица 9.11. Члены типа Color

Член	Назначение
FromArgb()	Возвращает объект типа Color. Для этого метода указываются числовые значения прозрачности, красного, зеленого и синего цветов
FromKnownColor()	Возвращает объект типа Color. Используются значения из перечисления KnownColor
FromName()	Возвращает объект типа Color. Используются строковые значения (например, Red)
A, **R, G, B**	Эти свойства возвращают значения, присвоенные параметрам прозрачности (A), красного (R), зеленого (G) и синего (B) цветов
IsNamedColor()**Name**	Эти члены применяются к текущему объекту Color. Они позволяют определить, соответствует ли он какому-либо из именованных цветов (например, Red), и, если соответствует, позволяют вернуть имя цвета
GetBrightness() GetHue() GetSaturation()	Помимо самой распространенной системы RGB существует и другая система цветовоспроизведения — HSB (Hue-Saturation-Brightness, оттенок–насыщенность–яркость). Эта методы позволяют получать для текущего объекта Color соответственно значения яркости, оттенка и насыщенности
ToArgb()	Возвращает числовые значения ARGB для объекта Color
ToKnownColor()	Возвращает значение из перечисления KnownColor для объекта Color

Возможности класса ColorDialog

В пространстве имен System.Windows.Forms предусмотрен заранее готовый класс ColorDialog, который обеспечивает пользователей приложения диалоговым окном для выбора цвета (рис. 9.7). Обратите внимание, что значения RGB или HSB могут быть выбраны как при помощи цветового поля, так и путем введения известных числовых значений.

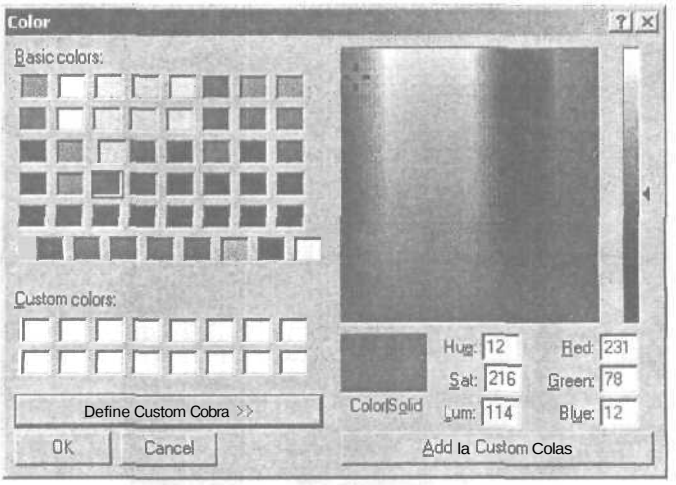


Рис. 9.7. Стандартное диалоговое окно для выбора цвета, которое можно использовать в приложении

Работа с классом ColorDialog производится очень просто. Первое, что нужно сделать - создать объект этого класса в программе. Затем, чтобы открыть диалоговое окно для пользователя, воспользуемся методом ShowDialog(). После того как пользователь сделает выбор, получить объект Color (соответствующий сделанному выбору) можно с помощью свойства ColorDialog.Color.

Например, предположим, что мы хотим обеспечить пользователю возможность выбирать при помощи этого диалогового окна цвет фона главной формы приложения. Чтобы упростить нашу задачу, сделаем так, чтобы диалоговое окно выбора цвета открывалось по щелчку мыши в любом месте клиентской части формы. Решение может выглядеть так:

```
public class ColorDlgForm : System.Windows.Forms.Form
{
        // Работаем с классом ColorDialog
        private System.Windows.Forms.ColorDialog colorDlg;
        public ColorDlgForm()
        {
                colorDlg = new System.Windows.Forms.ColorDialog();
                Text = "Click on me to change the color";
                this.MouseUp += new MouseEventHandler(this.ColorDlgForm_MouseUp);
        }
...
        private void ColorDlgForm_MouseUp(object sender, MouseEvent Args e)
        {
                if (colorDlg.ShowDialog() != DialogResult.Cancel)
                {
                        currColor = colorDlg.Color;
                        this.BackColor = currColor;
                        // Выводим информацию о выбранном цвете
                        string strARGB = colorDlg.Color.ToString();
                        MessageBox.Show(strARGB, "Color is:");
                }
        }
}
```

Результат работы этого приложения представлен на рис. 9.8.

Рис. 9.8. Работаем с классом ColorDialog

Несмотря на то что мы еще не обсуждали работу с диалоговыми окнами, предыдущий код вряд ли должен вызвать у нас какие-либо затруднения. Обратите внимание, что мы можем определить, какую кнопку нажал пользователь в диалоговом окне (OK или Cancel), проверяя значение, возвращаемое методом ShowDialog(), на предмет соответствия значениям перечисления DIalogResuH. В этой главе мы будем работать с другими диалоговыми окнами, а создавать свои собственные научимся в главе 10.

Код приложения ColorDlg можно найти в подкаталоге Chapter 9.

Работа со шрифтами

Главный класс, который используется для работы со шрифтами в GDI+, — это класс System.Drawing.Font. Объекты этого класса представляют конкретные шрифты, установленные на компьютере. В этом классе предусмотрено множество перегруженных конструкторов, но вот два наиболее часто используемых варианта:

```
// Создаем объект Font, указывая имя шрифта и его размер
Font f = new Font("Times New Roman", 12);

// Создаем объект Font, указывая имя, размер и стиль
Font f2 - new Font("WingDings", 50. FontStyle.Bold | FontStyle.Underline);
```

При создании f2 мы использовали стили из перечисления FontStyle. Как мы видим, можно задавать сразу несколько стилей одновременно. Значения из перечисления FontStyle представлены в табл. 9.12.

Таблица 9.12. Доступные стили из перечисления FontStyle

Член перечисления FontStyle	Стиль
Bold	Полужирный
Italic	Курсив
Regular	Обычный текст
Strikeout	Зачеркнутый
Underline	Подчеркнутый

После того как мы настроили необходимые параметры объекта Font, наша следующая задача — передать их методу Graphics.DrawString(). Несмотря на то что этот метод многократно перегружен, как правило, приходится указывать стандартный набор информации: текстовую строку, которая будет выводиться, используемые шрифт и кисть (толщина линии), а также область вывода. Например:

```
// public void DrawString(String, Font, Brush, Point);
g.DrawString("My string", new Font("Pop", 25). new SolidBrush(Color.Black),
                                                    new Point (0,0));

// public void DrawString(String, Font. Brush, float, float);
g.DrawString("Another string", new Font("Times New Roman". 16),
                                    new SolidBrush(Color.Red), 40. 40);
```

В обоих примерах мы использовали тип SolidBrush (с конкретным цветом). Вполне возможно одновременно использовать несколько типов кистей. В нашей ситуа-

ции нас вполне устраивает сплошная кисть (solid brush), более экзотические разновидности логических кистей будут рассмотрены ниже в этой главе.

После того как объект Font создан, мы можем использовать его многочисленные свойства, такие как Bold, Italic, Unit, Height, Size, FontFamily и многие другие.

Семейства шрифтов

В пространстве имен System.Drawing определен еще один важный тип — FontFamily, семейство шрифтов. Этот тип определяет наборы гарнитур, схожие по начертанию, но отличающиеся стилем или размером. Например, семейство шрифтов Verdana включает в себя шрифты «Verdana размер 12 полужирный» и «Verdana размер 24 курсив».

Конструктор типа FontFamily принимает текстовое имя семейства шрифтов. После того как объект FontFamily создан, его можно использовать для получения конкретного шрифта:

```
// Создаем объект FontFamily
FontFamily myFamily - new FontFamily("Verdana");

// Передаем созданный нами объект как параметр для конструктора Font
Font myFont = new Font(myFamily, 12);
e.Graphics.DrawString("Hello?", myFont, Brushes.Blue, 10, 10);
```

Для разработчиков приложений, в которых важную роль играет правильное размещение шрифта в отведенной ему области, большой интерес представляют члены типа FontFamily. Например, при помощи них можно определить среднюю ширину символа данного шрифта, выяснить, насколько могут выступать символы этого шрифта над строкой или под строку и т. п. Наиболее важные члены FontFamily представлены в табл. 9.13. Обратите внимание, что каждый из этих методов принимает только значение из перечисления FontStyle.

Таблица 9.13. Члены типа FontFamily

Член	Назначение
GetCellAscent()	Возвращает значение верхнего выступа для шрифтов-членов этого семейства
GetCellDescent()	Возвращает значение нижнего выступа для шрифтов-членов этого семейства
GetEmHeight()	Возвращает квадрат, определяемый самой широкой и самой высокой буквами шрифта (значение Em)
GetLineSpacing()	Возвращает расстояние между двумя соседними строками шрифта для шрифтов данного семейства при указанном FontStyle
GetName()	Возвращает имя семейства шрифтов
IsStyleAvailable()	Определяет, будет ли указанный FontStyle доступен для шрифтов из данного семейства

В качестве примера приведем обработчик события Paint, который возвращает набор характеристик семейства шрифтов Verdana:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
        Graphics g = e.Graphics;

        FontFamily myFamily = new FontFamily("Verdana");
```

```
        Font myFont = new Font(myFamily, 12);

        int y = 0;                          // Эта переменная будет использоваться у нас
                                            // для смещения по координате y
        int fontHeight - myFont.Height;     // Получаем высоту шрифта в пикселах

        // Выводим информацию о единицах измерения, используемых для членов FontFamily
        this.Text = "Measurements are in GraphicsUnit." + myFont.Unit.ToString();

        g.DrawString("The Verdana family.", myFont, Brushes.Blue, 10, y);
        y += 20;

        // Выводим всю информацию о нашем семействе шрифтов
        g.DrawString("Ascent for bold Verdana: " +
                     myFamily.GetCellAscent(FontStyle.Bold),
                     myFont, Brushes.Black, 10, y + fontHeight);
        y += 20;

        g.DrawString("Descent for bold Verdana; " +
                     myFamily.GetCell Descent(FontStyle.Bold),
                     myFont. Brushes.Black, 10, y + fontHeight);
        y += 20;

        g.DrawString("Line spacing for bold Verdana: " +
                     myFamily.GetLineSpaci ng(FontStyle.Bold),
                     myFont, Brushes.Black, 10, y + fontHeight);
        y += 20;

        g.DrawString("Height for bold Verdana: " + myFamily.GetEmHeight(FontStyle.Bold),
                     myFont. Brushes.Black, 10, y + fontHeight);
        y += 20;
    }
```

Результат выполнения этой программы представлен на рис. 9.9. Обратите внимание, что члены FontFaml ly возвращают значения в пунктах (Point), которые равны 1/72 дюйма, а не в пикселах. Если есть необходимость, мы можем задать вывод этих значений в других единицах измерений.

Рис. 9.9. Как получить параметры шрифта при помощи объекта FontFamily

Код приложения FontFamily можно найти в подкаталоге Chapter 9.

Единицы измерения для шрифта

Если еще вы никогда не работали на таком «типографском» уровне со шрифтами, полезно будет ознакомиться с некоторыми терминами из этой области. **Все** разме-

ры шрифта основываются на двух воображаемых линиях, которым соответствуют верхняя и нижняя границы символов данного шрифта. Некоторые символы (такие как j, y, g) выступают за пределы нижней границы, другие (b, f, h) — за пределы верхней. Высота шрифта — это расстояние между самой нижней точкой символа, который более всего выступает вниз, и самой верхней точкой символа, который более всего выступает вверх (рис. 9.10).

Рис. 9.10. Анатомия шрифта

Создаем приложение с возможностью выбора шрифта

Чтобы лучше освоить работу со шрифтами, мы создадим приложение, которое позволит пользователю выбирать нужный ему тип шрифта при помощи меню Configure (Настроить) ▸ Font Face (Гарнитура шрифта). То, что должно получиться, представлено на рис. 9.11.

Рис. 9.11. Приложение с возможностью выбора шрифта

Кроме того, чтобы было интереснее, мы разрешим пользователю косвенно управлять размером шрифта. Пусть при выборе пользователем пункта меню Configure (Настроить) ▸ Swell? (Разбухать?) шрифт начнет увеличиваться скачками через равные промежутки времени (при помощи объекта Timer), пока не достигнет установленного нами верхнего предела.

Первое, что мы должны сделать, — создать новый класс, производный от System.Winows.Forms.Form (представляющий главную форму нашего приложения). Затем мы должны позаботиться о переменных: для объекта Timer, для хранения информации о гарнитуре шрифта и для текущего размера шрифта:

```
public class FontForm: System.Windows.Forms.Form
{
        private Timer timer;
        private int swellValue;
        // Этот шрифт будет использоваться по умолчанию:
```

```
        private string fontFace = "WingDings";

        public FontForm()
        {
                // Считаем, что система меню создана при помощи графических средств Visual
                // Studio.IDE
                InitializeComponents();

                timer = new Timer();
                Text = "Font App";
                Width = 425;
                Height = 150;
                BackColor = Color.Honeydew;
                CenterToScreen();

                // Настраиваем таймер
                timer.Enabled = true;
                timer.Interval = 100;
                timer.Tick += new EventHandler(FontForm_OnTimer);
        }
}
```

Для подключения главного меню наш конструктор вызывает метод InitializeComponents(). Со средствами создания главного меню мы познакомились в предыдущей главе, поэтому если у вас возникли какие-либо вопросы относительно системы меню, советуем вам обратиться к исходному коду приложения (приложение FontApp в подкаталоге Chapter 9).

С объектом Timer мы также познакомились в предыдущей главе. Обработчик события Tick должен увеличивать значение переменной swellValue и перерисовывать клиентскую область нашей формы. Чтобы шрифт при «разбухании» не превысил все разумные размеры, мы установим для swellvalue максимум, равный 50. И еще один момент: чтобы уменьшить мерцание, связанное с перерисовкой всей клиентской части формы, мы будем обновлять только определенный «грязный» прямоугольник на форме:

```
private void FontForm_OnTimer(object sender. EventArgs e)
{
        // При каждом "тике" размер шрифта увеличивается на 5
        swellValue += 5;

        // При достижении максимального размера уменьшаем размер до нуля
        if(swellValue >= 50)
                swellValue = 0;

        // Перерисовываем нужную нам прямоугольную область формы
        Invalidate(new Rectangle(0, 0. ClientRectangle.Width, 100));
}
```

Теперь примерно каждые 100 миллисекунд значение swellValue будет изменяться, а верхние 100 пикселов клиентской части формы — перерисовываться. Нам осталось связать swellValue и размер шрифта, а также обеспечить вывод текстовой строки на форму. Выглядеть это может так:

```
private void FontForm_Paint(object sender. PaintEventsArgs e)
{
```

```
        Graphics g = e.Graphics;

        // Размер шрифта будет меняться между 12 и 62 в зависимости от текущего значения
        // swellValue
        Font theFont = new Font(fontFace, 12 + swellValue);

        string message = "Hello GDI+";

        // Выводим сообщение по центру формы
        float windowCenter = this.DisplayRectangle.Width/2;
        SizeF stringSize = g.MeasureString(message, theFont);
        float startPos = windowCenter - (stringSize.Width/2);

        g.DrawString(message, theFont, new SolidBrush(Color.Blue), startPos, 10);
    }
```

Последнее, что мы должны сделать, — настроить включение и отключение «разбухания» шрифта при выборе пользователем пункта меню Configure (Настроить) ▸ Swell? (Разбухать?). Для этого необходимо настроить обработчик ConfigSwell_Clicked для включения и отключения объекта Timer;

```
    private void ConfigSwell_Clicked(object sender, EventArgs e)
    {
        timer.Enabled = !timer.Enabled;
        mainMenu.MenuItems[1].MenuItems[0].Checked = timer.Enabled;
    }
```

Выводим информацию об установленных шрифтах (System.Drawing.Text)

Давайте наделим наше приложение FontApp дополнительными возможностями. Пусть это приложение будет программным образом получать информацию об установленных в системе шрифтах и выводить эту информацию на форму. Для этой цели нам потребуются типы из еще одного пространства имен GDI+ — System.Drawing.Text. Самые интересные для нас типы из этого пространства имен представлены в табл. 9.14.

Таблица 9.14. Типы пространства имен System.Drawing.Text

Тип	Назначение
InstalledFontCollection	Представляет набор всех шрифтов, установленных на компьютере
PrivateFontCollection	Коллекция для объектов Font
LineSpacing	*Это* перечисление определяет значения, при помощи которых можно настроить расстояние между строками текста
TextRenderingHint	Еще одно перечисление, при помощи которого можно установить качество графики при выводе текста. Например, при использовании значения Text текст будет выводиться очень быстро, но с невысоким качеством изображения, при использовании AntiAliased качество изображения возрастет за счет скорости вывода и т. п.

Вывод списка установленных в системе шрифтов будет производиться при выборе пользователем пункта меню Configure (Настроить) ▸ List Installed Fonts (Вывести установленные шрифты) — рис. 9.12.

Рис. 9.12. Еще один пункт в меню Configure (Настроить)

Механизм работы будет таким: при выборе пользователем этого пункта меню активируется соответствующий обработчик события (mnuConfigShowFonts_Clicked()). Первое, что делает этот метод — создает объект класса InstalledFontCollection. Этот класс содержит массив FontFamily, представляющий набор всех шрифтов, установленных на данном компьютере. Получить доступ к объектам этого массива можно при помощи свойства InstalledFontCollection.Families. Последнее, что нам осталось сделать,— извлечь имя каждой гарнитуры шрифта при помощи свойства FontFamily.Name. Полученные значения записываются в переменную типа string, которую мы назовем installedFonts:

```
public class FontForm : System.Windows.Forms.Form
{
        // Для хранения списка шрифтов
        private string installedFonts;

        // Обработчик события меню для вывода списка установленных в системе шрифтов
        private void mnuConfigShowFonts_Clicked(object sender, EventArgs e)
        {
                InstalledFontCollection fonts = new InstalledFontCollection();

                for(int i=0; i < fonts.Families.Lenght; i++)
                {
                        installedFonts += fonts.Families[i].Name + " ";
                }

                // На этот раз нам потребуется перерисовать всю клиентскую область формы,
                // поскольку вывод полученных значений будет производиться в нижнюю часть
                // формы
                Invalidate();
        }
...
}
```

Последнее, что мы должны сделать, — вывести значение переменной installedFonts на форму. При этом нам необходимо учесть, что в верхнюю часть формы уже производится вывод «разбухающей» строки Hello GDI+, и сделать так, чтобы выводимые строки не мешали друг другу:

```
private void FontForm_Paint(object sender. PaintEventArgs e)
{
        Graphics g - g.Graphics;
        Font theFont = new Font(fontFace, 12 + swellValue);
        string message = "Hello GDI+";

        // Выводим сообщение "Hello GDI+" по центру форны
        float windowCenter = this.DisplayRectangle.Width/2;
        SizeF stringSize - g.MeasureString(message, theFont);
        float startPos - windowCenter - (stringSize.Width/2);
```

```
        g.DrawString(message, theFont, new SolidBrush(Color.Blue), startPos, 10);

        // Выводим информацию об установленных шрифтах под занятой областью
        Rectangle myRect = new Rectangle(0, 100, ClientRectangle.Width,
                                         ClientRectangle.Height);

        // Будем рисовать в этой области белым по черному
        g.FillRectangle(newSolidBrush(Color.Black), myRect);
        g.DrawString(installedFonts, new Font("Arial", 12), new SSolidBrush(Color.White),
                                                                myRect);
}
```

Вспомним, что «грязный прямоугольник» для неспокойной надписи "Hello GDI+" ограничивается верхними 100 пикселами клиентской части формы. Поскольку по событию Tick перерисовывается только эта часть формы, нижняя часть (в которую выводится список установленных шрифтов) остается в неприкосновенности. В результате форма мерцает меньше.

И для порядка сделаем так, чтобы при возникновении события Resize (то есть при изменении размера формы пользователем) нижняя часть формы перерисовывалась правильно:

```
private void FontForm_Resize(object sender, System.EventArgs e)
{
        Rectangle myRect = new Rectangle(0, 100, ClientRectangle.Width,
                                                 ClientRectangle.Height);
        Invalidate(myRect);
```

То, что должно получиться в итоге, представлено на рис. 9.13.

Код приложения FontApp можно найти в подкаталоге Chapter 9.

Рис. 9.13. Вывод информации об установленных в системе шрифтах

Класс FontDialog

Как вы, наверное, уже догадались, класс FontDialog предназначен для вывода диалогового окна, в котором пользователь сможет выбрать нужный ему шрифт и установить требуемые параметры (рис. 9.14).

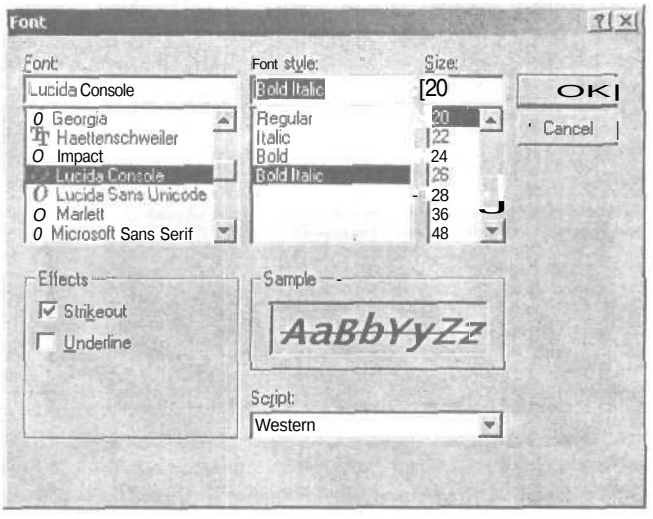


Рис. 9.14. «Полуфабрикатное» диалоговое окно Font

Ранее в этой главе мы уже рассматривали полуфабрикатное диалоговое окно ColorDialog. Работа со всеми «полуфабрикатными» диалоговыми окнами очень похожа. Чтобы вызвать на экран окно FontDialog, необходимо точно так же воспользоваться методом ShowDialog(). Затем для получения информации о том, что пользователь выбрал, следует воспользоваться свойством Font. В качестве примера представим форму, которая будет работать с окном FontDialog (рис. 9.15) точно так же, как ранее созданная нами форма с окном для выбора цвета (то есть окно для выбора шрифта будет открываться при щелчке мышью в любой точке клиентской части формы):

```
public class FontDlgForm : System.Windows.Forms.Form
{
        private System.Windows.Forms.FontDialog fontDlg:
        private Font currFont;
        // Обработчик события Paint
        private void FontDlgForm_Paint(object sender. PaintEventArgs e)
        {
                Graphics g = e.Graphics;
                g.DrawString("Testing...", currFont. new SolidBrush(Color.Black). 0. 0);
        }
        public FontDlgForm()
        {
                CenterToScreen();
                fontDlg = new System.Windows.Forms.FontDialog();
                fontDlg.ShowHelp = true;
                Text = "Click on me to change the font";
                currFont = new Font("Times New Roman". 12):
                ...
        }

        // Обработчик события MouseUp
        private void FontDlgForm_MouseUp(object sender. MouseEventArgs e)
        {
```

```
        if (fontDlg.ShowDialog() != DialogResult.Cancel)
        {
            currFont = fontDlg.Font;
            Invalidate();
        }
    }
}
```

Рис. 9.15. Работаем с диалоговым окном FontDialog

Код приложения FontDlgForm можно найти в подкаталоге Chapter 9.

Обзор пространства имен System.Drawing.Drawing2D

Мы уже много раз использовали в наших примерах типы Pen (перо) и Brush (кисть). Самые простые варианты этих типов (которыми мы обычно и обходились) определены в пространстве имен System.Drawing. Однако есть гораздо более интересные перья и кисти, и для их использования (так же как и множества других возможностей) нам необходимо обратиться к пространству имен System.Drawing.Drawing2D.

Это дополнительное пространство имен GDI+ (со значительно меньшим количеством типов, чем System.Drawing) обеспечивает возможность устанавливать специальные «наконечники» для перьев (pen caps), создавать кисти, которые рисуют не сплошной полосой, а текстурами, производить различные векторные манипуляции с графическими объектами. Некоторые наиболее интересные типы этого пространства имен представлены в табл. 9.15.

Таблица 9.15. Классы System.Drawing.Drawing2D

Класс	Назначение
AdjustableArrowCap CustomLineCap	Определяют «наконечники» для перьев. В результате исходная точка и конец линии получают характерные завершения. Можно использовать выбранный пользователем вариант: или стрелки на конце линии (AdjustableArrowCap), или полностью определенное им самим завершение (CustomLineCap)
Blend ColorBlend	Используются для смешивания цветов. Обычно используются вместе с LinearGradientBrush
GraphicsPath GraphicsPathIterator	Объект GraphicsPath представляет набор связанных линий (прямых и кривых). В данный объект можно поместить практически любой тип геометрической фигуры (например, дуги, прямоугольники, отрезки прямой линии, многоугольники и т. п.)

Класс	Назначение
PathData	Хранит графические данные для GraphicsPath
HatchBrush LinearGradientBrush PathGradientBrush	Экзотические типы кистей

Для этих типов используются значения из перечислений, также определенных в пространстве имен System.Drawing.Drawing2D. Эти перечисления представлены в табл. 9.16.

Таблица 9.16. Перечисления System.Drawing.Drawing2D

Перечисление	Назначение
DashStyle	Определяет стиль штриховых линий для пера
FillMode	Определяет заполнение внутренней области геометрической фигуры
HatchStyle	Определяет варианты штриховки (для объектов HatchBrush)
LinearGradientMode	Определяет направление градиентного изменения цвета
LineCap	Определяет стиль «наконечника» пера
PenAlignment	Определяет ориентацию пера относительно проводимой им линии
PenType	Определяет тип линии, создаваемой пером
QualityMode SmoothingMode RenderingHint	Определяют качество вывода графического объекта

Определение качества вывода графического объекта

Такие перечисления из пространства имен System.Drawing.Drawing2D, как QualityMode и SmoothingMode, определяют значения, при помощи которых можно задать качество вывода какого-либо графического объекта. Для любого объекта Graphics определено качество вывода по умолчанию — как правило, оно представляет собой компромисс между качеством вывода и скоростью выполнения графической операции. Однако иногда возникает необходимость отказаться от значения по умолчанию в пользу большего выигрыша по скорости или по качеству вывода.

Перечисление SmoothingMode (табл. 9.17) обычно используется для более точной настройки качества вывода графического объекта с точки зрения применения к нему технологии сглаживания (antialiasing).

Чтобы выбрать для графического объекта один из возможных режимов сглаживания, следует использовать свойство SmoothingMode объекта Graphics:

```
private void MainForm_Paint(object sender. PaintEventArgs e)
{
        Graphics g = e.Graphics;

        // Устанавливаем качество вывода графического объекта
        g.SmoothingMode = SmoothingMode.AntiAlias;
        ...
}
```

Таблица 9.17. Возможные значения SmoothingMode

Значение	Описание
AntiAlias	Определяет вывод со сглаживанием неровностей. Обычно это означает, что к краям линий применяются полутона, которые позволяют получить эффект сглаживания (без «зазубрин»). Изображение будет одинакового хорошо выглядеть как на электронно-лучевых, так и на жидкокристаллических мониторах
HighQuality	Еще более высокое качество за счет снижения скорости вывода. В этом режиме используются достаточно изощренные технологии, например, позволяющие применять возможности субпиксельного разрешения на жидкокристаллических мониторах. При этом отдельный пиксел разбивается на три подпиксела, для каждого из которых вычисляется отдельное значение полутонов таким образом, чтобы скругления линий выглядели идеально плавными при зрительном восприятии
Highspeed	Наибольшая скорость вывода за счет снижения качества. Полутона не используются

Свойство SmoothingMode может использоваться только для управления качеством вывода графических объектов, но не текста, Для того чтобы настроить нужное нам качество для вывода текста (через объект Font), используется свойство TextRenderingHint, для которого устанавливаются значения из перечисления System.Drawing.TextRenderingHint.

Работа с перьями

Обычное применение объектов Pen (перьев) заключается в рисовании линий. Как правило, объект Pen используется не сам по себе: он передается в качестве параметра многочисленным методам вывода, определенным в классе Graphics. Как правило, названия всех этих методов, использующих Pen, начинаются с Draw (схожие по функциональности методы, принимающие объект Brush (кисть), начинаются на Fill, но об этом позже).

Со многими из методов DrawXXXX() мы уже встречались на протяжении этой главы. В табл. 9.18 они сведены воедино и охарактеризованы более подробно. Большинство из этих методов многократно перегружены.

Таблица 9.18. Методы DrawXXXX() класса Graphics, принимающие в качестве параметра объект Pen

Метод	Назначение
DrawArc()	Этот метод предназначен для вывода дуги. Он принимает в качестве параметров объект Pen и данные, позволяющие построить эллипс и выделить на нем дугу
DrawBezier() DrawBeziers()	Метод для вывода кубической кривой Безье (нескольких кривых) по четырем точкам
DrawCurve()	Метод для вывода кривой на основе массива точек
DrawEllipse()	Метод для вывода эллипса, вписанного в прямоугольник (передаются координаты прямоугольника)
DrawLine() DrawLines()	Эти методы соединяют прямыми линиями точки (массив точек)
DrawPath()	Этот метод выводит коллекцию прямых и кривых линий при помощи типа GraphicsPath, определенном в пространстве имен System.Drawing.Drawing2D

Метод	Назначение
DrawPie()	Выводит часть эллипса, заключенную между дугой эллипса и двумя радиальными линиями
DrawPolygon()	Выводит многоугольник на основе принимаемого массива точек
DrawRectangle() DrawRectangles()	Выводят прямоугольник (или несколько прямоугольников), основываясь на двух точках: для верхнего левого и нижнего правого угла. Можно передавать как объект Rectangle, так и координаты точек в виде значений int и float

Теперь, когда мы получили представление о том, в каких ситуациях используются объекты класса Реп, мы рассмотрим возможности этого класса. В нем предусмотрено несколько перегруженных конструкторов, при помощи которых можно задать исходный цвет и толщину пера (объект Реп можно также создать на основе существующего объекта Brush, но об этом позже). Большая часть возможностей Pen определяется свойствами этого класса. Перечень наиболее важных свойств представлен в табл. 9.19.

Таблица **9.19.** Свойства класса Реп

Свойство	Назначение
Brush	Определяет кисть, используемую данным объектом Реп
Color	Определяет цвет создаваемых объектом Реп линий
CompoundArray	Позволяет получить или создать массив пользовательских вариантов штрихов и пустого пространства между штрихами
CustomStartCap CustomEndCap	Позволяют получить или установить стиль «наконечника» пера, который будет показан в начале линии (StartCap) и в конце линии (EndCap)
DashCap	Позволяет получить или установить стиль «наконечника» для перьев, рисующих пунктирные линии
DashOffset	Устанавливает смещение начала пунктира относительно исходной точки пунктирной линии
DashPattern	Позволяет получить или установить массив штрихов и пробелов между ними для пунктирных линий
DashStyle	Позволяет получить или установить стиль для пунктирных линий, создаваемых при помощи данного объекта Реп
LineJoin	Позволяет получить или установить стиль объединения при пересечении двух линий, выводимых данным объектом Реп
PenType	Позволяет получить стиль линий, выводимых при помощи данного объекта Реп
StartCap EndCap	Позволяет получить или установить один из заранее готовых стилей «наконечника» пера. Используются значения из перечисления LineCap, определенного в пространстве имен System.Drawing.Drawing2D
Width	Позволяет получить или установить ширину данного пера

Кроме класса Реп в GDI+ также можно использовать коллекцию заранее определенных перьев (коллекция Pens). При помощи статических свойств коллекции Pens мы можем мгновенно получить уже готовое перо, без необходимости создавать его вручную. Однако все типы Реп, которые создаются при помощи коллекции Pens, имеют одну и ту же одинаковую ширину, равную 1. Изменить ее (начиная с версии Visual Studio Beta 2) мы уже не сможем.

Для того чтобы проиллюстрировать приведенную выше информацию, мы выведем несколько геометрических фигур с использованием класса Pen. Будем считать, что главная форма у нас уже создана. Вывод графических изображений при помощи класса Pen на нее будет производиться следующим образом:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
        Graphics g = e.Graphics;

        // Создаем большое перо синего цвета
        Pen bluePen - new Pen(Color.Blue, 20);

        // Создаем еще одно перо при помощи заготовок из коллекции Pens
        Pen pen2 = Pens.Firebrick;

        // Выводим при помощи созданных нами перьев геометрические фигуры
        g.DrawEllipse(bluePen, 10, 10, 100, 100);
        g.DrawLine(pen2, 10, 130, 110, 130);
        g.DrawPie(Pens.Black, 150, 10, 120, 150, 90, 80);

        // Выводим многоугольник пурпурного цвета
        Pen pen3 = new Pen(Color.Purple, 5);
        pen3.DashStyle = DashStyle.DashDotDot;

        g.DrawPolygon(pen3, new Point[] { new Point (30, 140),
                                          new Point(265, 200),
                                          new Point(100, 225),
                                          new Po1nt(l90, 190),
                                          new Point(50, 330),
                                          new Point(20, 180),} );
        // Добавляем прямоугольник со вписанным нами текстои
        Rectangle r = new Rectangle(150, 10, 130, 60);
        g.DrawRectangle(Pens.Blue, r);
        g.DrawString("Hello out there...How are ya?",
                     new Font("Arial", 12), Brushes.Black, r);
}
```

Получившаяся в итоге причудливая форма представлена на рис. 9.16.

Обратите внимание, что для объекта Pen, использованного для вывода многоугольника, было применено значение из перечисления DashStyle (определенного в System.Drawing.Drawing2D). Это перечисление заменило в .NET флаги стиля пера в Win32 (PS_SOLID). Значения из перечисления DashStyle представлены в табл. 9.20.

Таблица 9.20. Значения перечисления DashStyle

Значение	Перечисление
Custom	Пользовательский стиль пунктире
Dash	Штриховая линия
DashDot	Штрихпунктирная линия: штрих -- точка — штрих
DashDotDot	Штрихпунктирная линия: штрих — точка — точка — штрих (как в нашем примере)
Dot	Пунктир из одних точек
Solid	Сплошная линия

Помимо использования готовых стилей пунктирных линий из перечисления DashStyles, мы можем также определить свой собственный стиль. Для этого используется свойство DashPattern класса Pen:

```
// Выводим пунктирную линию нашего собственного стиля по периметру формы
Pen customdashPen = new Pen(Color.BlueViolet, 5);
float[] myDashes = {5.0f, 2.0f, 1.0f, 3.0f};

customDashPen.DashPattern = myDashes;
g.DrawRectangle(customDashPen, ClientRectangle);
```

Результат выполнения обновленной программы (обратите внимание на рамку вокруг формы) представлен на рис. 9.17.

Код приложения РепApp можно найти в подкаталоге Chapter 9.

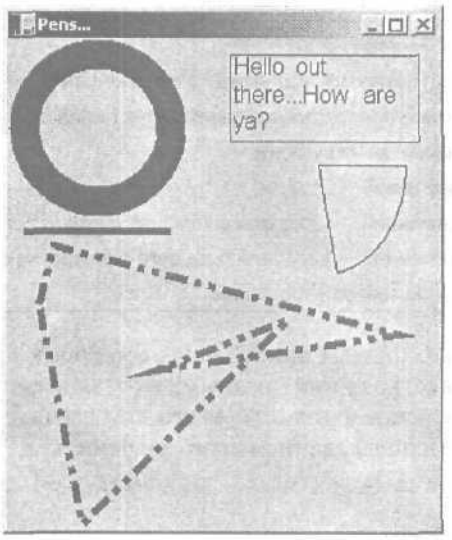


Рис. 9.16. Возможности класса Реп

Рис. 9.17. Используем свой собственный стиль пунктирной линии

Работаем с «наконечниками» перьев

При рассмотрении предыдущей формы можно обнаружить, что концы всех нарисованных на ней линий обрезаны стандартным образом — под прямым углом. Если мы предпочитаем более изысканные окончания линий, в нашем распоряжении — перечисление LineCap. Значения этого перечисления представлены в табл. 9.21.

Таблица 9.21. Значения перечисления LineCap

Значение	Описание
ArrowAnchor	Линии оканчиваются стрелками
DiamondAnchor	Линии оканчиваются «бриллиантами» (ромбами)
Flat	Стандартное прямоугольное завершение линий
Round	Линии на концах скруглены
RoundAnchor	На концах линий — «шары»
Square	На концах линий — квадраты в толщину линии
SquareAnchor	На концах линий — квадраты большего размера, чем толщина линии
Triangle	Треугольное завершение линий

В качестве примера создадим приложение, в котором при помощи класса Pen будет нарисовано множество линий с разными окончаниями. Наш код выведет для каждого из значений перечисления LineCap его имя (типа ArrowAnchor), а затем нарисует рядом линию с использованием этого значения:

```
private void MainForm_Paint (object sender, PaintEventArgs e)
{
        Graphics g = e.Graphics;
        Pen thePen = new Pen(Color.Black, 10);
        int yOffSet = 10;

        // Получаем все значения перечисления LineCap
        Array obj = Enum.GetValues(typeof(LineCap));

        // Выводим линию с использованием значения из LineCap
        for(int x = 0; x < obj.Length; x++)
        {
                // Настраиваем "наконечник" пера
                LineCap temp = (LineCap)obj.GetValue(x);
                thePen.StartCap = temp;
                thePen.EndCap = temp;

                // Выводим имя значения перечисления
                g.DrawString(temp.ToString(), new Font("Times New Roman", 10),
                                        new SolidBrush(Color.Black), 0, yOffSet);

                // Выводим линию с выбранным наконечником
                g.DrawLine(thePen, 100, yOffSet, Width - 50, yOffSet);

                yOffSet += 40;
        }
}
```

То, что должно получиться, представлено на рис. 9.18.

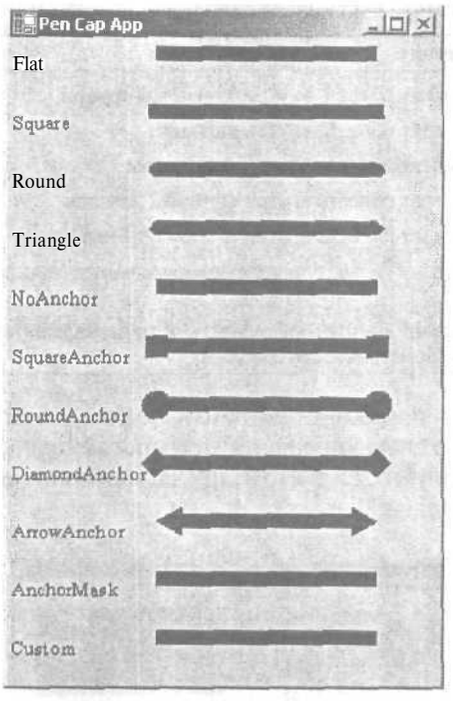


Рис. 9.18. «Наконечники» для перьев

Код приложения PenCapApp можно найти в подкаталоге Chapter 9.

Работаем с кистью

Будем считать, что к этому моменту мы уже свободно владеем пером. Следующий инструмент мастера GDI+ — это кисть (brush). Кисти предназначены для «закрашивания» пространства между линиями. Мы можем определить для кисти цвет, текстуру или даже изображение. Сам класс Brush является абстрактным, и создавать объекты этого класса мы не можем. Вместо этого в нашем распоряжении классы, производные от Brush, такие как SolidBrush, HatchBrush, LinearGradientBrush и т. п, Кроме того, создавать объекты кистей (выбрав из заранее готового набора) можно при помощи типов-коллекций Brushes и System.Brushes, также определенных в пространстве имен System.Drawing. Создание объектов из этих типов-коллекции производится при помощи их статических свойств. Далее мы можем передать созданный объект кисти в качестве параметра соответствующему методу объекта Graphics. В предыдущем примере мы передавали объект класса SolidBrush методу Graphics.DrawString() для вывода текстовых строк. Однако «родные» методы, которые принимают объекты кистей, это — методы FillXXXX(). Набор этих методов представлен в табл. 9.22.

Кроме того, выбранную кисть также можно использовать для создания объекта Pen (перо). Это перо сможет рисовать линии, используя все возможности кисти (например, линия будет покрыта текстурами или изображением).

Таблица 9.22. Методы RIIXXXX() класса Graphics

Метод	Назначение
FillClosedCurve()	Закрашивает область внутри замкнутой кривой
FillEllipse()	Закрашивает область внутри эллипса
FillPath()	Закрашивает область внутри траектории
FillPie()	Закрашивает область внутри сегмента эллипса
FillPolygon()	Закрашивает область внутри многоугольника
FillRectangle() FillRectangles()	Закрашивают область внутри прямоугольника (нескольких прямоугольников)
FillRegion()	Закрашивает внутреннюю область объекта Region (Region — это внутренняя область геометрической фигуры)

В качестве примера приведем программу, которая будет использовать типы SolidBrush и Brushes для закрашивания пространства внутри геометрических фигур. То, что должно получиться в итоге, представлено на рис. 9.19 (выглядит немного знакомо, не правда ли?).

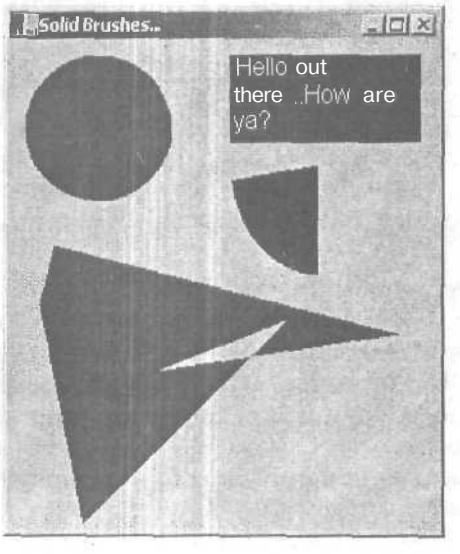


Рис. 9.19. Работаем с объектами кистей

Конечно же, это вариант нашего приложения PenApp, в котором мы решили закрасить внутренние пространства геометрических фигур при помощи кистей SolidBrushes и методов FillXXXX() вместо перьев Реп и методов DrawXXXX(). Выглядеть соответствующий код может так:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
        Graphics g = e.Graphics;

        // Создаем "плотную кисть" - SolidBrush - синего цвета
        SolidBrush blueBrush = new SolidBrush(Color.Blue);
```

```
        // Создаем еще одну кисть при помощи заранее готовой коллекции Brushes
        SolidBrush pen2 = (SolidBrush)Brushes.Firebrick;

        // Закрашиваем этими кистями геометрические фигуры
        g.FillEllipse(blueBrush, 10, 10, 100, 100);
        g.FillPie(Brushes.Black, 150, 10, 120, 150, 90, 80);

        // Закрашиваем многоугольник пурпурным цветом
        SolidBrush brush3 = new SolidBrush(Color.Purple);

        g.FillPolygon(brush3, new Point[]{ new Point(30, 140),
                                           new Point(265, 200),
                                           new Point(100, 225),
                                           new Point(190, 190),
                                           new Point(50, 330),
                                           new Point(20, 180)} );
        // и прямоугольник с текстом - синим:
        Rectangle r = new Rectangle(150, 10, 130, 60);
        g.FillRectangle(Brushes.Blue, r);
        g.DrawString("Hello out there...Now are ya?",
               new Font("Arial", 12), Brushes.White, r);
    }
}
```

Код приложения SolidBrushApp можно найти в подкаталоге Chapter 9.

Работаем со штриховыми кистями

Более сложное «закрашивание» можно произвести при помощи производного от Brush класса HatchBrush, определенного в пространстве имен System.Drawing.Drawing2D. Этот тип позволяет закрасить внутреннюю область объекта при помощи большого количества шаблонов штриховки, определенных в перечислении HatchStyle. Этих шаблонов действительно очень много, поэтому в табл. 9.23 мы приводим только некоторые из них.

Таблица 9.23. Значения перечисления HatchStyle (стили штриховки)

Значение	Описание
BackwardDiagonal	Диагональная штриховка с наклоном вправо
Crass	«Крестообразная» штриховка, состоящая из пересекающихся вертикальных и горизонтальных линий
DiagonalCross	Еще одна разновидность «крестообразной» штриховки, состоящая из пересекающихся диагональных линий
Forward Diagonal	Диагональная штриховка с наклоном влево
Hollow	«Пустая» кисть, которая ничего не рисует
Horizontal	Горизонтальная штриховка
Pattern	Штриховка, которая создается на основе указанного пользователем растрового изображения
Solid	Обычная «плотная» кисть без всякой штриховки (аналогично обычному типу SolidBrush)
Vertical	Вертикальная штриховка

При создании объекта HatchBrush нам обязательно нужно будет указать два цвета: цвет «переднего плана» и цвет фона. В качестве примера мы внесем изменения

в наше приложение PenCapApp. Теперь оно вместо линий с завершениями будет выводить эллипсы со штриховкой:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
        Graphics g = e.Graphics;
        int yOffSet - 10;

        // Помещаем в массив все члены перечисления HatchStyle
        Array obj = Enum.GetValues(typeof(HatchStyle));

        // Выводим эллипс со штриховкой, соответствующей членам перечисления HatchStyle
        // с 1 по 10
        for(int x -0; x < 10; x++)
        {
                // Настраиваем кисть
                HatchStyle temp = (HatchStyle)obj.GetValue(x);
                HatchBrush theBrush - new HatchBrush(temp, Color.White, Color.Black);

                // Выводим имя каждого из значений перечисления
                g.DrawString(temp.ToString(), new Font("Times New Roman", 10),
                                              new SolidBrush (Color.Black), 0. yOffSet);

                // Закрашиваем эллипс штриховой кистью
                g.FillEllipse(theBrush, 150, yOffSet. 200, 25);
                yOffSet += 40;
        }
}
```

Результат работы программы представлен на рис. 9.20.
Код приложения BrushStyles можно найти в подкаталоге Chapter 9.

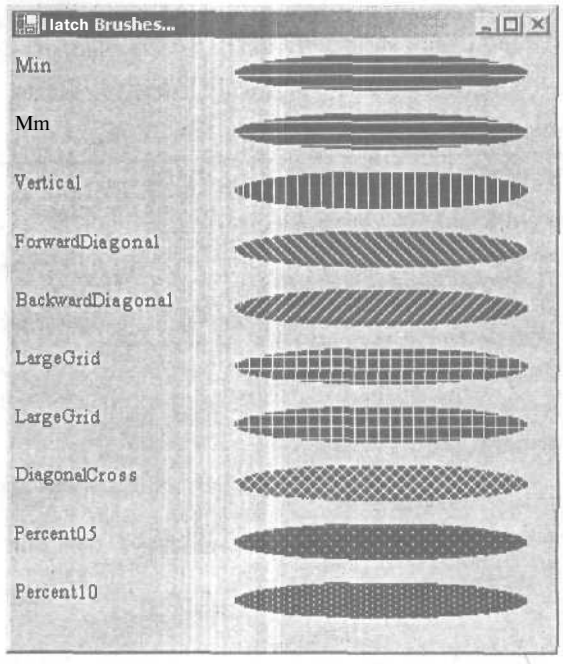


Рис. 9.20. Стили штриховки

Работаем с текстурными кистями

Помимо штриховых кистей, в распоряжении разработчика GDI+ имеются также текстурные кисти, представленные типом TextureBrush. Эти кисти «закрашивают» отведенную для этого область текстурой — то есть указанным нами растровым изображением. Об особенностях работы с изображениями в GDI+ (при помощи класса Image) вы узнаете несколькими страницами позже, а пока мы скажем только, что для TextureBrush используется ссылка на объект Image, представляющий изображение. Изображение может быть внешним файлом (в формате *.bmp, *.gif или *.jpg) или сборкой .NET.

Как обычно, мы проиллюстрируем применение класса TextureBrush на примере. В нашем случае этот класс будет использован для того, чтобы:

- залить всю клиентскую площадь формы текстурой из файла clouds.bmp (изображением облаков);
- залить выводимый на форме текст текстурой на основе файла soap bubbles.bmp (изображением мыльных пузырей).

То, что должно получиться, представлено на рис. 9.21.

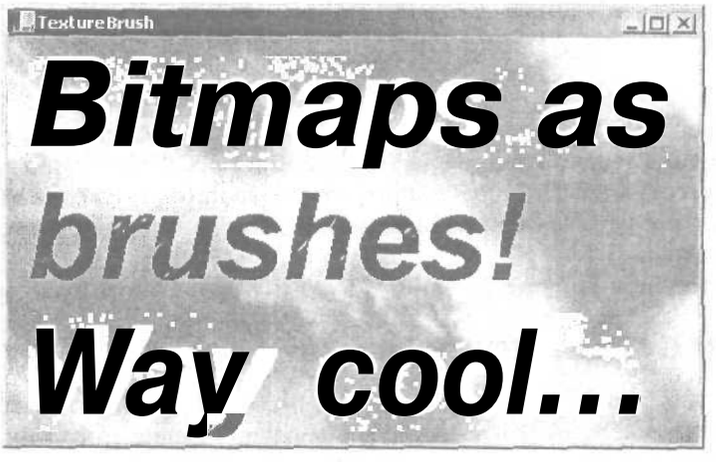


Рис. 9.21. Применение текстурных кистей

Сам код приложения очень прост. В первую очередь (в конструкторе для формы) мы создаем два объекта TextureBrush. Обратите внимание, что конструктор TextureBrush требует ссылки на объект Image:

```
public class MainForm : System.Windows.Forms.Form
{
        // Нам потребуются эти переменные типа Brush для загрузки изображений
        private Brush texturedTextBrush;
        private Vrush texturedBGroundBrush;

        public MainForm()
        {
                ...
                // Загружаем изображение для текстуры формы
```

```
            Image bGroundBrushImage = new Bitmap(Coluds.bmp");
            texturedBGrounBrush = new TextureBrush(bBroundBrushImage);

            // Загружаем изображение для текстуры теста
            Image textBrushImage = new Bitmap("Soap Bubbles.bmp");
            texturedTextBrush = new TextureBrush(textBrushImage);
        }
...
}
```

Теперь в нашем распоряжении есть два полностью готовых к употреблению объекта TextureBrush. Осталось привязать их к форме и тексту на форме соответственно:

```
private void MainForm_Paint(object sender. PaintEventArgs e)
{
        Graphics g - e.Graphics;
        Rectangle r = ClientRectangle;

        // Выводим облака на форме
        g.FillRectangle(texturedBGroundBrush, r);

        // Текст должен быть больший - чтобы можно было разглядеть текстуру
        g.DrawString ("Bitmaps as brushes! Way cool...",
                        new Font("Arial", 60. FontStyle.Bold | FontStyle.Italic),
                        texturedTextBrush.  r);
}
```

По сравнению с усилиями, которые потребуются, чтобы достичь того же эффекта с помощью только «голого» API или даже MFC, C# обеспечивает исключительную простоту и эффективность при работе с достаточно сложными эффектами. Однако у нас осталась еще одна весьма впечатляющая разновидность кисти: градиентная кисть.

Код приложения TexturedBrush можно найти в подкаталоге Chapter 9.

Работаем с градиентными кистями

Последний тип кисти, который мы рассмотрим, — это градиентная кисть, представленная типом LinearGradientBrush. Основное назначение этого типа — обеспечить плавное смешение двух цветов для получения градиентного перехода. Работать с этим типом так же просто, как и с ранее рассмотренными типами кистей. Единственная особенность использования этого типа заключается в том, что мы должны указать направление цветового перехода при помощи значений из перечисления LinearGradientMode (табд. 9.24).

Таблица 9.24. Значения перечисления LinearGradientMode

Значение	*Описание* (направление перехода)
BackwardDiagonal	Из верхнего правого угла в нижний левый
ForwardDiagonal	Из верхнего левого угла в нижний правый
Horizontal	Слева направо
Vertical	Сверху вниз

Чтобы посмотреть на каждое из значений LinearGradientMode в действии, мы воспользуемся уже знакомым нам подходом, нарисовав на форме серию прямоугольников при помощи LinearGradientBrush. В качестве цветов, между которыми будет происходить переход, выберем Color.Red и Color.Blue:

```
private void MainForm_Paint(object sender. PaintEventArgs e)
{
        Graphics g = e.Graphics;
        Rectangle r = new Rectangle(10, 10, 100, 100);

        LinearGradientBrush theBrush - null;
        int yOffSet = 10;

        // Получаем все значения перечисления LinearGradientMode
        Array obj = Enum.GetValues(typeof(LinearGradientMode));

        // Выводим прямоугольник, используя значения LinearGradientMode
        for(int x - 0; x < obj.Length; x++)
        {
                // Настраиваем кисть
                LinearGradientMode temp - (LinearGradientMode)obj.GetValue(x);
                theBrush - new LinearGradientBrush(r, Color.Red, Color.Blue, temp);

                // Выводим имя значения перечисления
                g.DrawString(temp.ToString(), new Font("Times New Roman", 10).
                                        new SolidBrush(Color.Black), 0. yOffSet);

                // Заполняем прямоугольник при помощи градиентной кисти
                g.FillRectangle(theBrush, 150. yOffSet, 200. 50);
                yOffSet += 80;
        }
}
```

Результат выполнения программы представлен на рис. 9.22.

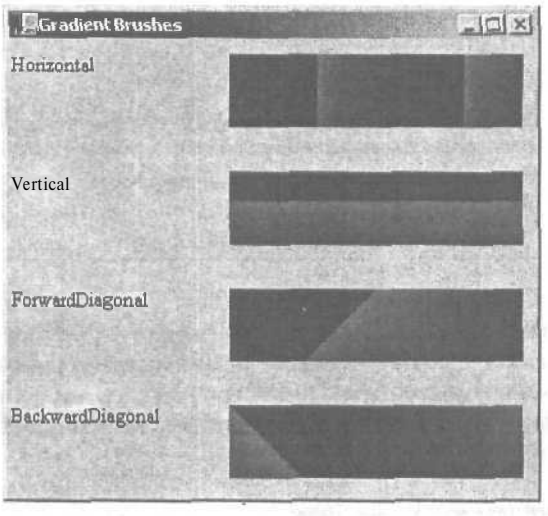


Рис. 9.22. Применение градиентных кистей

Код приложения GradientBrush можно найти в подкаталоге Chapter 9.

Вывод изображений

В предыдущих разделах **мы** уже познакомились с применением трех из четырех главных типов GDI+ (шрифтов, перьев и кистей). Нам осталось рассмотреть последний из основных типов: тип System.Drawing.Image, который используется для вывода изображений. Класс Image определяет множество свойств и методов, которые можно использовать для настройки параметров выводимого изображения. К примеру, при помощи свойств Width, Height и Size можно получить или установить размеры изображения.

Кроме того, в пространстве имен System.Drawing.Imaging определено множество типов для проведения сложных преобразований изображений. Если бы мы поставили задачу рассмотреть эти типы более или менее подробно, нам бы, пожалуй, потребовалась отдельная книга. Поэтому мы сосредоточимся только на наиболее часто используемых возможностях работы с изображениями.

Наиболее важные члены класса Image представлены в табл. 9.25. Многие из этих членов являются статическими, а некоторые — абстрактными.

Таблица 9.25. Члены класса Image

Член	Назначение
FromFile()	Этот статический метод предназначен для создания объекта Image из файла
FromHbitmap()	Еще один статический метод. Создает объект Bitmap на основе идентификатора окна (Window handle)
FromStream()	Этот статический метод позволяет создать объект Image, используя в качестве источника поток данных
Height Width Size Physical Dimensions HorizontalDimensions Vertical Resolution	Все эти свойства предназначены для работы с размерами (измерениями) изображения
Palette	Это свойство возвращает объект ColorPalette, представляющий цветовую палитру, использованную для данного графического изображения
GetBounds()	Возвращает прямоугольник, представляющий текущую область, занятую изображением
Save()	Позволяет сохранить изображение в файл

Класс Image является абстрактным, и создавать объекты этого класса нельзя, Обычно объявленные переменные Image присваиваются объектам класса Bitmap. Кроме того, мы можем создавать объекты класса Bitmap напрямую. Например, предположим, что нам необходимо вывести на форму три изображения. Мы можем объявить три переменные Image, а затем использовать для каждой из них объекты Bl tmap:

```
public class mainForm : System.Windows.Forms.Form
{
        // Объявляем переменные типа Image
        private Image bMapImageA;
        private Image bMapImageB;
        private Image bMapImageC;
```

```
        public MainForm()
        {
            ...
            // Используем для этих переменных объекты класса Bitmap
            bMapImageA = new Bitmap("ImageA.bmp");
            bMapImageB = new Bitmap("ImageB.bmp");
            bMapImageC = new Bitmap("ImageC.bmp");
        }
...
}
```

Вывод полученных изображений производится очень просто, поскольку в классе Graphics предусмотрен специальный метод, который так и называется — DrawImage(). Этот метод многократно перегружен, поэтому в нашем распоряжении множество вариантов того, как поместить изображение в нужное нам место на форме. Кроме того, для настройки параметров выводимого изображения мы можем использовать с этим методом значения перечислений ImageAttributes и GraphicsUnit. В нашей ситуации нам нужно всего лишь указать место, на которое будут помещены изображения, чтобы они не перекрывали друг друга. Координаты можно указать при помощи объектов Point (точка), Rectangle (прямоугольник), целочисленными значениями или значениями с плавающей запятой. В нашем случае решение может быть таким:

```
protected void OnPaint (object sender. System.Windows.Forms.PaintEventArgs e)
i
        Graphics g - e.Graphics;

        // Выводим изображения
        g.DrawImage(bMapImageA, 10, 10, 90. 90);
        g.DrawImage(bMapImageB, 10. 110. 90. 90);
        g.DrawImage(bMapImageC, 10, 210. 90. 90);
}
```

То, что должно получиться, представлено на рис. 9.23.

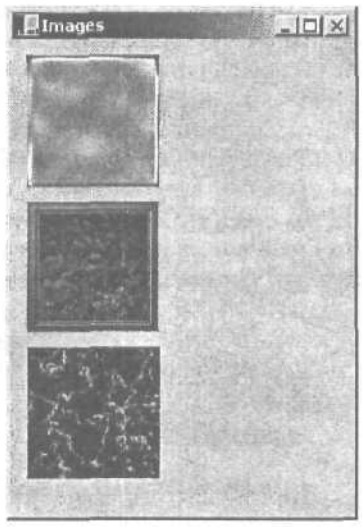


Рис. 9.23. Вывод изображений

Класс Bitmap позволяет выводить изображения, которые хранятся в файлах самого разного формата. Например:

```
// Тип Bitmap поддерживает все распространенные форматы!
Bitmap myBMP = new Bitmap("CoffeeCup.bmp");
Bitmap myGIF = new Bitmap("Candy.gif");
Bitmap myJPEG - new Bitmap("Clock.jpg");
Bitmap myPNG = new Bitmap("Speakers.png");
Bitmap myTIFF = new Bitmap("FooFighters.bmp");

// Выводим изображения при помощи Graphics.DrawImage()
g.DrawImage(myBmp,  10.  10);
g.DrawImage(myGIF,  220.  10);
g.DrawImage(myJPEG,  280.  10);
g.DrawImage(myPNG,  150.  200);
g.DrawImage(myTIFF,  300.  200);
```

Код приложения Images можно найти в подкаталоге Chapter 9.

Перетаскивание, проверка попадания в область, занимаемую изображением, и элемент управления PictureBox

Как мы уже убедились, можно без проблем выводить изображение прямо на область, занимаемую формой или другим элементом управления. Однако в нашем распоряжении будет гораздо больше возможностей, если мы поместим изображение на форму не напрямую, а при помощи специального элемента управления PictureBox. Что же мы получим в результате? Очень многое. PictureBox — это класс, производный от Control, поэтому мы сможем захватывать множество событий, использовать всплывающую подсказку, контекстное меню и прочие возможности элемента управления. Чтобы обеспечить все эти возможности для изображения без использования PictureBox, нам придется приложить немало усилий.

Проиллюстрируем применение PictureBox на примере. Мы создадим приложение, которое будет захватывать события MouseUp, MouseDown и MouseMove для области, занятой на форме изображением (которое помещено в PictureBox). Для того чтобы было интереснее, мы добавим код для работы с перетаскиванием (drag-and-drop). Пользователь сможет перетаскивать одно из изображений по всей форме. Кроме того, мы будем осуществлять проверку того, где оказалось перетаскиваемое изображение после того, как пользователь его отпустит. Если оно попало в область, занимаемую заданным нами прямоугольником, будет выводиться специальное сообщение. Таким образом мы реализуем механизм, называемый «проверкой попадания» — hit testing.

Почти все возможности PictureBox наследуются от уже знакомого нам класса Control. Назначение объекту PictureBox изображения, которое будет в нем содержаться, выглядит следующим образом:

```
public class MainForm : System.Windows.Forms.Form
!
        // Наш PictureBox будет содержать изображение улыбающейся рожицы
        private PictureBox happyBox;

        public MainForm()
```

```
{
        // Настраиваем параметры объекта PictureBox
        happyBox = new PictureBox();
        happyBox.SizeMode = PictureBoxSizeMode.StretchImage;
        happyBox.Location = new System.Drawing.Point(64, 32);
        happyBox.Size = new System.Drawing.Size(50, 50);
        happyBox.Cursor - Cursors.Hand;

        happyBox.Image = new Bitmap("happy.bmp");

        // Добавляем объект PictureBox в коллекцию Controls формы:
        Controls.Add(happyBox);
    }
    ...
}
```

Единственное свойство PictureBox, которое может вызвать какие-либо вопросы — это свойство SizeMode, для которого используются значения из перечисления PictureBoxSizeMode. Это свойство позволяет определить, как именно будет выводиться изображение внутри PictureBox. В нашем случае было использовано значение StretchImage, которое означает, что изображение должно быть сжато или растянуто таким образом, чтобы полностью соответствовать размерам PictureBox. Другие возможные значения представлены в табл. 9.26.

Таблица 9.26. Значения перечисления PictureBoxSizeMode

Значение	Описание
AutoSize	Значения PictureBox будут изменены таким образом, чтобы полностью соответствовать размерам изображения
CenterImage	Если размеры PictureBox будут больше, чем размеры изображения, изображение будет позиционировано точно по центру PictureBox, Если же размеры изображения будут превышать размеры PictureBox, то выступающие края будут обрезаны
Normal	Изображение будет расположено в верхнем левом углу PictureBox. Если размеры изображения превысят размеры PictureBox, выступающие края будут обрезаны

После того как мы создали PictureBox и настроили его свойства, следующая наша задача — обеспечить обработчики для событий MouseMove, MouseUp и MouseDown. Поскольку PictureBox — это класс, производный от Control, работа с этими событиями производится точно так же, как и в случае других элементов управления:

```
// Добавляем обработчики для следующих событий
happyBox.MouseDown += new MouseEventHandler(happyBox_MouseDown);
happyBox.MouseUp += new MouseEventHandler(happyBox_MouseUp);
happyBox.MouseMove += new MouseEventHandler(happyBox_MouseMove);
```

В обработчике события MouseDown, во-первых, устанавливается значение true для переменной isDragging (это будет означать, что началась операция перетаскивания), а во-вторых, фиксируются координаты указателя мыши при наступлении этого события:

```
// Обработчик события MouseDown для объекта PictureBox
private void happyBox_MouseDown(object sender, MouseEventArgs e)
{
    isDragging - true;
```

```
        // Сохраняем координаты (x. y) для исходного положения указателя мыши.
        // Они понадобятся нам для расчета смещения PictureBox
        oldX = e.X;
        oldY = e.Y:
}
```

Обработчик события MouseMove обеспечивает перемещение PictureBox по форме (изменяя значения свойств Top и Left). Для расчета нового положения PictureBox используется смещение указателя мыши относительно исходной позиции:

```
// Если пользователь производит щелчок на изображении и, не отпуская кнопку, перемещает
// мышь, PictureBox вместе с изображением будет перерисовываться в новом месте
private void happyBox_MouseMove(object sender. MouseEventArgs e)
{
        if(isDragging)
        {
                // Определяем новое значение координаты Y для PictureBox по разности между
                // старым и новым положением указателя мыши
                happyBox.Top = happyBox.Top + (e.Y - oldY);

                // То же самое для координаты X
                happyBox.Left = happyBox.Left + (e.X - oldX);
        }
}
```

Далее обработчик события MouseUp должен установить значение переменной isDragging равным false — это будет индикатор окончания операции перетаскивания. Кроме того, по условиям задачи должна производиться проверка: если перемещаемый PictureBox отпущен внутри определенной области формы (ограниченной размерами объекта Rectangle — прямоугольника), то будем считать, что пользователь добился успеха. Таким образом, весь оставшийся необходимый код выглядит следующим образом:

```
// Когда пользователь отпустит кнопку мыши, операция перетаскивания завершится.
// Проверяем, не оказалось ли при этом перетаскиваемое изображение внутри заданного
// прямоугольника:
private void happyBox_MouseUp(object sender. MouseEventArgs e)
{
        isDragging = false;

        // Находится ли указатель мыши внутри "прямоугольника сбрасывания" - dropRect?
        if(dropRect.Contains(happyBox.Bounds))
        {
                MessageBox.Show("You win!". "What an amazing test of skill...");
        }
}
// Будем считать. что на форме уже выведен прямоугольник следующих размеров:
// Rectangle dropRect "- new Rectangle(100, 100. 150. 150);
//
private void MainForm_Paint(object sender, PaintEventArgs e)
{
        // Выводим "прямоугольник сбрасывания"
        Graphics g = e.Graphics;
        g.FillRectangle(Brushes.AntiqueWhite, dropRect);

        // Инструкции пользователю
        g.DrawString("Drag the happy guy in here...", new Font("Times New Roman". 25).
                                                        Brushes.Red, dropRect);
}
```

Как мы видим, главную роль в проверке попадания играет метод Rectangle.Contains(). Этот метод перегружен и может принимать другой прямоугольник (объект Rectangle), точку (объект Point) или два значения типа int в качестве координат. Этот метод исключительно удобен в ситуации, когда нам необходимо проверить, попал ли пользователь щелчком мыши в прямоугольную область на форме или нет.

Результат работы нашей программы представлен на рис. 9.24. Если пользователь соблазнится возможностями нашей игры и сумеет в ней победить, то в качестве награды его ждет окно сообщения, представленное на рис. 9.25.

Рис. 9.24. Перетаскивание графических объектов и проверка попадания

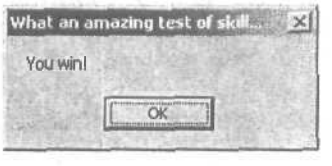


Рис. 9.25. Настоящая проверка мастерства

Код приложения DraggingImages можно найти в подкаталоге Chapter 9.

Еще о проверке попадания

Реализовать проверку попадания для любого типа, производного от Control, очень просто, поскольку в Control предусмотрен набор событий мыши. Однако как обеспечить проверку попадания не на элемент управления, а в другое место формы, например внутрь границ геометрической фигуры? Давайте изменим наше предыдущее приложение Images таким образом, чтобы реализовать проверку попадания без использования возможностей типа Control.

Какмы помним, в нашем приложении Images на форму было выведено три изображения. При этом ни одно из этих изображений *не заключено* в PictureBox (что упростило бы задачу). Наша задача — при щелчке пользователя разобраться, в какой именно области формы был произведен этот щелчок, и если щелчок пришелся на какое-либо из изображений, заключить это изображение в красную рамку и изменить заголовок формы (это делается при помощи свойства Form.Text). То, что должно получиться, представлено на рис. 9.26.

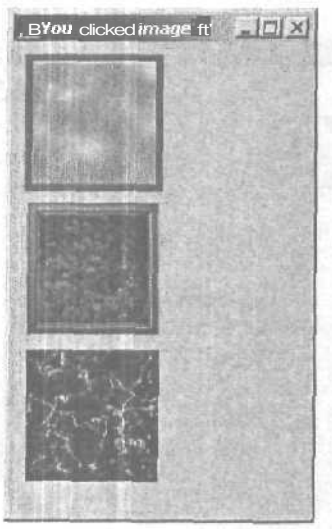


Рис. 9.26. Выделение изображений

Первая наша задача - обеспечить перехват события MouseDown для самой формы. После этого мы сможем захватывать координаты указателя мыши в момент щелчка и осуществлять проверку — попадает или нет указатель мыши в область, занимаемую одним из изображений. Если ответ положительный, то значения двух переменных изменятся: переменной isImageClicked (типа bool) будет присвоено значение true, а переменной imageCli cited (типа int) - значение, соответствующее номеру выбранного пользователем изображения. Выглядеть все это может так:

```
public class MainForm : System.Windows.Forms.Form
{
...
        // Две переменные
        private bool isImageClicked - false:
        private int ImageClicked;

        protected void OnMouseDown (object sender. MouseEventArgs e)
        {
                // Получаем координаты указателя мыши в момент щелчка
                Point mousePt = new Point(e.X, e.Y);

                // Проверяем, не попадает ли указатель мыши в одну из трех областей,
                // занимаемых изображениями .
```

```
            if(rectA.Contains(mousePt))
            {
                  isImageClicked = true;
                  imageClicked = 0;
                  this.Text = "You clicked image A";
            }
            else if(rectB.Contains(mousePt))
            {
                  isImageClicked = true;
                  imageClicked = 1;
                  this.Text = "You clicked image B";
            }
            else if(rectC.Contains(mousePt))
            {
                  isImageClicked = true;
                  imagedicked = 2;
                  this.Text = "You clicked image C";
            }
            // Пользователь не попал в изображения, устанавливаем значения
            // по умолчанию
            else
            {
                  isImageClicked = false;
                  this.Text = "Images";
            }

            // Перерисовываем форму
            Invalidate();
      }
...
}
```

Обратите внимание, что в наше приложение добавлена еще одна проверка: для щелчка мыши, который не попадает ни в одно из трех изображений. Эта проверка нужна для удаления рамки с выбранного перед этим изображения.

После того как все нужные нам значения переменных установлены, мы перерисовываем клиентскую область формы. Вот обработчик для события Paint:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
      Graphics g = e.Graphics;

      // Выводим все три изображения
      ...

      // Выводим рамку (по щелчку пользователя)
      if(isImageClicked == true)
      {
            Pen outline = new Pen(Color.Red, 5);

            switch(imageClicked)
            {
                  case 0:
                        g.DrawRectangle(outline, rectA);
                        break;
                  case 1:
                        g.DrawRectangle(outline, rectB);
```

```
                    break;
                case 2:
                    g.DrawRectangle(outline, rectC);
                    break;
                default:
                break;
            }
        }
    }
```

Проверка попадания в непрямоугольные области

В предыдущих ситуациях наша задача была относительно простой: нам необходимо было реализовать проверку попадания в прямоугольные области. Однако как быть, есть нам нужно сделать то же самое для более сложной геометрической фигуры? Например, предположим, что нам потребовалось по щелчку пользователя выделять красным контуром фигуру, представленную справа на рис. 9.27.

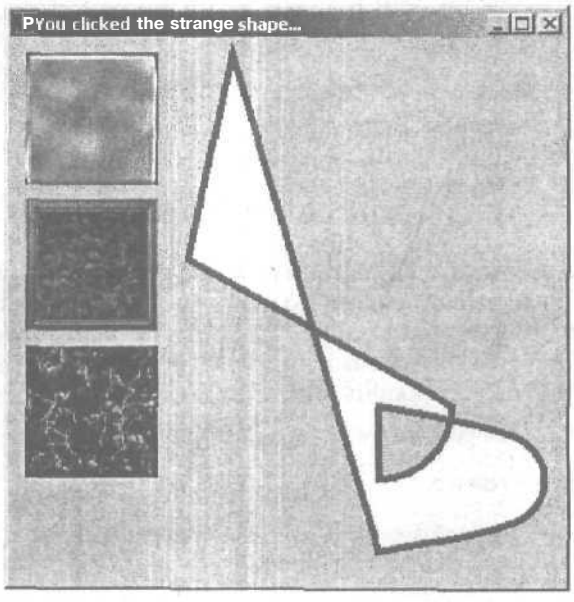


Рис. 9.27. Как реализовать проверку попадания для фигуры с причудливыми очертаниями?

Странная фигура, представленная на рис. 9.27, была выведена на форму при помощи метода `Graphics.FillPath()`. Этот метод принимает экземпляр объекта `GraphicsPath`, о котором уже упоминалось ранее при рассмотрении пространства имен `System.Drawing.Drawing2D`. Объект `GraphicsPath` инкапсулирует набор связанных прямых и кривых линий, строк и т. п. Добавление новых элементов в объект `GraphicsPath` производится при помощи многочисленных методов `Add`, представленных в табл. 9.27.

Таблица 9.27. Методы Add класса GraphicsPath

Метод	Назначение
AddArc()	**Добавляет в создаваемую** фигуру **дугу — сегмент эллипса**
AddBezier() AddBeziers()	**Добавляют кривую** Безье **(или нескольких кривых Безье)**
AddClosedCurve()	**Добавляет замкнутую кривую**
AddCurve()	**Добавляет кривую**
AddEllipse()	**Добавляет эллипс**
AddLine() AddLines()	**Добавляют прямую линию (несколько прямых линий)**
AddPath()	**Добавляет** указанный **вами объект GraphicsPath**
AddPie()	**Добавляет сегмент эллипса,** образованный **дугой эллипса** и двумя **радиальными линиями**
AddPolygon()	**Добавляет многоугольник**
AddRectangle() AddRectangles()	**Добавляют прямоугольник (несколько прямоугольников)**
AddString()	**Добавляет текстовую** строку

Давайте добавим в наше приложение Images объект GraphicsPath. Добавление в этот объект элементов будет производиться в конструкторе формы:

```
public MainForm : System.Windows.Forms.Form
{
        // Создаем объект GraphicsPath
        GraphicsPath myPath - new GraphicsPath();

        public MainForm()
        {
                // Добавляем в объект GraphicsPath элементы, из которых он будет
                //  состоять
                myPath.StartFigure();
                        myPath.AddLine(new Point(150, 10). new Point(120, 150));
                        myPath.AddArc(200, 200, 100, 100, 0. 90):
                        Point pointl = new PointC250. 250);
                        Point point2 = new Point(350. 275);
                        Point point3 = new Point(350, 325);
                        Point point4 - new Point(250, 350);
                        Point[] points = {point1, po1nt2, point3, point4};
                        myPath.AddCurve(points);
                myPath.CloseFigure();

        Г
}
```

Обратите внимание на вызовы методов StartFigure() и CloseFigure(). Метод StartFigure() сигнализирует о том, что мы начинаем процедуру добавления новых элементов в объект GraphicsPath, а метод CloseFigure() — об окончании этой процедуры (после чего мы можем приступить к созданию новой фигуры). Если создаваемая при помощи объекта GraphicsPath фигура состоит из непрерывных линий (как в нашем случае), то метод CloseFigureO соединяет начальную и конечную точку фигуры.

В классе GraphicsPath определено еще много замечательных членов, но мы сосредоточимся на реализации проверки попадания в область, занимаемую нашим сложным объектом. Получение координат указателя мыши производится без каких-либо проблем при помощи обработчика события MouseDown. А далее сравнить эти координаты с областью, занимаемой объектом GraphicsPath, можно при помощи метода GraphicsPathЛsVisible() (точно такой же метод можно использовать и для объекта Regi on):

```
protected void OnMouseDown (object sender, MouseEventArgs e)
{
        // Получаем координаты указателя мыши при щелчке
        Point mousePt = new PointCe.X. e.Y);
        ...
        else if(myPath.IsVisible(mousePt))
        {
                isImageClicked = true;
                imagedicked - 3;
                this.Text = "You clicked the strange shape...";
        }
```

i"

Последнее, что мы должны сделать, — внести изменения в обработчик события Paint:

```
private void MainForm_Paint(object sender. PaintEventArgs e)
{
        Graphics g = e.Graphics;
        ...

        // Выводим объект GraphicsPath
        g.FillPath(Brushes.AliceBlue, myPath);

        // Обводим этот объект рамкой (по щелчку)
        if(isImageClicked = = true)
        {
                Pen outline = new Pen(Color.red, 5);

                switch(imageClicked)
                {
                        ...
                        case 3;
                                g.DrawPath(outline, myPath);
                                break;
                        default:
                                break;
                }
        }
}
```

Код приложения Images можно найти в подкаталоге Chapter 9.

Работа с форматами ресурсов .NET

Во всех предыдущих примерах мы работали исключительно с ресурсами (например, изображениями), которые были помещены во внешние файлы. Например,

в приложении Images все три выводимых изображения загружались из файлов, находящихся в каталоге приложения:

```
// Загружаем изображения из файлов
bMapImageA  =  new  fiitmapt"imageA.bmp"):
bMapImageB - new Bitmap("imageB.bmp");
bMapImageC - new Bitmap("imageC.bmp");
```

При этом в каталоге приложения обязательно должны находиться три файла: imageA.bmp, imageB.bmp и imageC.bmp, как показано на рис. 9.28.

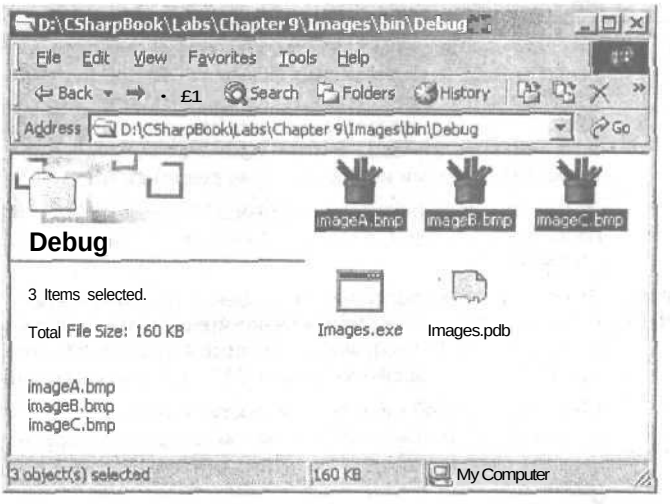


Рис. 9.28. Внешние ресурсы приложения

Если какой-либо из этих файлов будет удален, переименован или перемещен, наше приложение не будет работать (можете проверить сами). Если же поместить ресурсы, необходимые приложению, внутрь самой сборки, мы будем избавлены от подобных проблем.

Как мы помним (глава 6), сборка — это набор типов и *необязательныхресурсов*. Наша задача сейчас — выяснить, каким образом внешние ресурсы (такие как файлы изображений или текстовые строки) могут быть загружены непосредственно в саму сборку. Как правило, если мы собираемся делать это вручную, то нам придется выполнить следующие действия:

- создать файл *.resx в формате XML, в котором будут содержаться пары имя — значение для каждого из ресурсов;
- использовать утилиту resgen.exe для преобразования файла *.resx в формате XML в его двоичный эквивалент (файл *.resources);
- использовать для компилятора C# параметр /resource (или сокращенно /res) для вложения ресурсов непосредственно в сборку.

При использовании IDE Visual Studio.NET все эти операции выполняются автоматически. Однако мы останемся верны нашим традициям и вначале проделаем все операции вручную, а затем уже узнаем, как то же самое можно сделать при помощи Visual Studio.

Пространство имен System.Resources

Ключ к использованию форматов ресурсов .NET заключается в понимании роли типов пространства имен System.Resources. Эти типы позволяют работать как с файлами *.resx в формате XML, так и с двоичными файлами *.resources. Перечень наиболее важных типов представлен в табл. 9.28.

Таблица 9.28. Типы пространства имен System.Resources

Тип	Назначение
IResourceReader IResourceWriter	Эти интерфейсы реализуются типами, обеспечивающими чтение и запись ресурсов .NET (в самых разных форматах). Реализовывать эти интерфейсы самостоятельно стоит лишь в той ситуации, когда требуется создать свой собственный пользовательский класс для чтения и записи ресурсов. В большинстве случаев достаточно использовать типы, представленные ниже, в которых эти интерфейсы уже реализованы
ResourceReader ResourceWriter	Эти классы, реализующие интерфейсы IResourceReader и IResourceWriter, обеспечивают чтение и запись в двоичные файлы ресурсов .NET (*.resources)
ResXResourceReader ResXResourceWriter	Эти классы также реализуют интерфейсы IResourceReader и IResourceWriter. Они обеспечивают чтение и запись в файлы ресурсов в формате XML (*.resx). Файлы ресурсов в формате XML могут быть преобразованы в двоичные файлы .NET при помощи утилиты resgen.exe
ResourceManager	Обеспечивает удобный и простой доступ к ресурсам, зависящим от естественного языка (BLOB, текстовые строки) во время выполнения

Создание файлов *.resx программным образом

Файлы *,resx -- это файлы в формате XML, которые содержат ресурсы для нашей сборки. Ресурсы в них представлены в виде пар имя — значение. Создание файлов *.resx, добавление в них двоичных и текстовых ресурсов, а также их сохранение производится при помощи класса ResXResourceWriter. Проиллюстрируем его использование на примере.

Предположим, что наша задача — создать файл *.resx с двумя ресурсами: изображением happy.bmp, с которым мы уже работали на протяжении этой главы, и текстовой строкой. Графический интерфейс нашего приложения будет предельно простым (рис. 9.29).

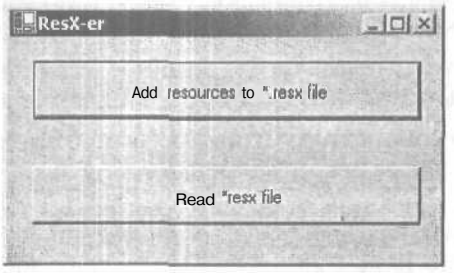


Рис. 9.29. Интерфейс пользователя для нашего приложения

Вся работа по записи ресурса в файл будет выполняться в обработчике события Click для верхней кнопки нашего приложения. Его код может выглядеть следующим образом:

```
protected void btnMakeResxFile_Click(object sender. System.EventArgs e)
{
        // Создаем объект ResXResourceWriter и указываем файл, в который будет
        // производиться запись
        ResXResourceWriter w = new ResXResourceWriter("ReXForm.resx");

        // Добавляем ресурс-изображение
        Image i = new Bitmap("happy.bmp");
        w.AddResource("happyDude", 1):

        // Добавляем ресурс-строку текста
        w.AddResource("welcomeString", "Hello new resource format!");

        // Сохраняем файл с добавленными ресурсами
        w.Generate();
        w.CloseC);
}
```

Обратите внимание на метод ResXResourceWriter.AddResource(). Этот метод несколько раз перегружен и позволяет добавлять как двоичные данные BLOB (в нашем случае графический файл happy.bmp), так и текстовые данные. Этот метод принимает два параметра: один — имя ресурса в файле *.resx, а второй — сам ресурс. Сохранение информации в файл производится при помощи метода ResXResourceWriter.Generate().

Как видим, нам совершенно нет необходимости брать на себя обязанности по написанию файла XML (с двоичными вставками) вручную — все это сделает за нас класс ResXResourceWriter. При этом действительно будет создан файл в формате XML: в этом несложно убедиться, открыв файл *.resx в Visual Studio. Делается это следующим образом: в Visual Studio воспользуемся командой Add Existing Item (добавить существующий элемент) в меню Project (Проект) и выберем нужный файл *.resx. Созданный нами файл ResXForm.resx будет выглядеть так, как показано на рис. 9.30.

Рис. 9.30. Представление ресурсов в формате XML

Мы легко сможем догадаться, где лежит happy.bmp, а где - текстовая строка, по именам ресурсов. Каждый из ресурсов в файле *.resx представлен в формате пары имя–значение, что значительно облегчает работу с ними.

Чтение из файлов *.resx программным образом

В нашем приложении есть еще одна кнопка. При нажатии на нее должно производиться чтение ресурсов из файла ResXForm.resx. Как вы уже, наверное, догадались, для этого используется класс ResXResourceReader. В нашем примере после открытия файла мы воспользуемся ссылкой на интерфейс IDictionaryEnumerator, а затем выведем данные для каждой пары имя — значение;

```
protected void btnReadResXFile_Click(object sender. System.EventArgs e)
{
        // Создаем объект ResXResourceReader
        ResXResourceReader r - new ResXResourceReader("ResXForm.resx");

        // Получаем ссылку на интерфейс IDictionaryEnumerator и используем ее для вывода
        // данных о всех ресурсах
        IDictionaryEnumerator en = r.GetEnumerator();

        while (en.MoveNext())
        {
                MessageBox.Show("Value:" + en.Value.ToString(). "Key: " +
                                                        en.Key.ToString());
        }
        r.Close();
}
```

При нажатии на кнопку мы увидим два окна сообщения с информацией о ресурсах внутри ResXForm.resx. На рис. Э.31 представлено сообщение, относящееся к картинке из happy.bmp.

Создание файлов *.resources

После того как мы создали файл *.resx в формате XML, следующая наша задача — преобразовать его в двоичный файл *.resources. Для этого предназначена утилита resgen.exe. Конечно же, Visual Studio.NET может выполнить эту операцию совершенно автоматически, но мы сейчас сделаем это вручную. Командная строка resgen.exe должна выглядеть следующим образом:

```
resgen rexform.resx rexform.resources
```

Если заглянуть в созданный таким образом файл resxform.resources, можно увидеть двоичный код ресурсов (рис. 9.32).

Встраивание файла ресурсов в сборку

Последнее, что нам осталось сделать, — поместить созданный нами файл *.resources в сборку. При использовании компилятора командной строки C# это можно сделать при помощи параметра /res. Не забудьте, что для csc.exe мы обязательно также должны указать все ссылки на внешние библиотеки;

```
csc /res:resxform.resources /r:System.Drawing.dll /r:System.Windows.Forms.dll /
                                                        r:System.dll resxform.cs
```

Открыв созданную сборку в ILDasm.exe, мы сможем увидеть в ее манифесте указание на вложенное хранилище ресурсов (рис. 9.33).

Чуть позже мы выясним, как можно во время выполнения обращаться к ресурсам, размещенным в сборке.

Код приложения ResXWriterReader можно найти в подкаталоге Chapter 9.

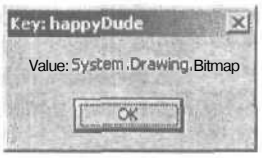


Рис. 9.31. Извлечение пары имя — значение

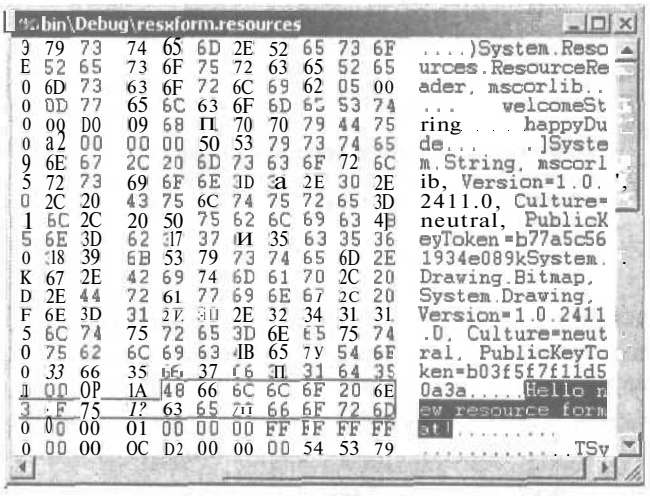


Рис. 9.32. Двоичный файл resxform.resources

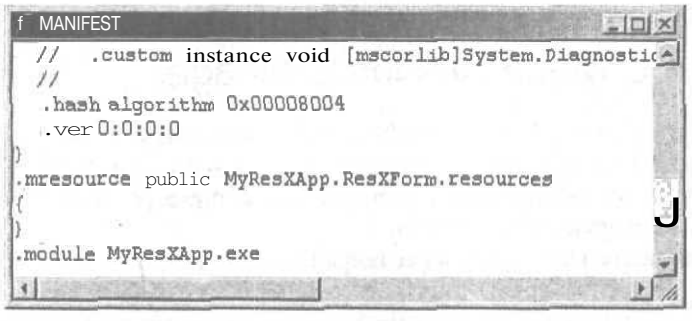


Рис. 9.33. Манифест сборки с ресурсами

Работаем с типом ResourceWriter

В предыдущих разделах мы использовали для записи и чтения ресурсов классы ResXResourceReader и ResXResourceWriter. Эти классы используются для выполнения

операций (чтения и записи) с файлами ресурсов *.resx в формате XML. После этого мы конвертировали эти файлы XML в двоичные файлы, а потом уже помещали полученные таким образом двоичные файлы в сборку.

Правда заключается в том, что мы вполне можем обойтись вообще без работы с файлами в формате XML. Для того чтобы создавать сразу двоичные файлы *.resources, вместо ResXResourceWriter следует использовать ResourceWriter, а для чтения их вместо ResXResourceReader — ResourceReader. В качестве примера мы создадим простое консольное приложение с использованием этих классов. Выглядеть оно будет следующим образом:

```
class ResourceGenerator
{
        static void Main(string[] args)
        {
                // Создаем новый файл *.resources
                ResourceWriter rw;
                rw = new ResourceWriter("myResources.resources");

                // Добавляем изображение и текстовую строку
                rw.AddResource("happyDude", new Bitmap("happy.bmp"));
                rw.AddResource("welcomeString", "Welcome to .NET resources.");
                rw.Generate;
        }
}
```

Чтобы создать файл *.resources, достаточно скомпилировать приложение и запустить его на выполнение. После этого мы сможем вложить полученный файл в сборку точно так же, как и раньше:

```
csc /res:myresources.resources /r:System.Drawing.dll /r:System.Windows.Forms.dll /
                                                      r:System.dll ResourcesGen.cs
```

Для считывания данных из файла *.resources используется класс ResourceReader. Его мы рассматривать на примере не будем, поскольку работа с ним производится точно так же, как и с классом ResXResourceWriter.

Работаем с типом ResourceManager

Скорее всего, в нашем приложении мы не будем использовать ни ResXResourceReader, ни ResourceReader. Причина проста — гораздо удобнее использовать класс ResourceManager. Этот тип позволяет извлекать двоичные и текстовые ресурсы непосредственно из сборки, в которую они вложены.

Давайте создадим в нашем проекте новый класс и назовем его MyResourceReader. Этот класс будет использовать тип ResourceManager для извлечения изображения happyDude и текста welcomeString и размещения их на форме: изображения — в PictureBox, а текста — в Label. Само извлечение ресурсов производится при помощи методов GetObject() и GetString(). Однако обратите внимание: все названия ресурсов, которые мы передаем этим методам в качестве параметров, чувствительны к регистру. Код MyResourceReader будет выглядеть так:

```
class MyResourceReader
{
```

```
    public void ReadMyResources()
    {
        // Открываем файл resources
        ResourceManager rm - new ResourceManager("myResources",
                                    Assembly.GetExecutingAssembly());

        // Загружаем ресурс-изображение
        PictureBox p = new PictureBox();
        Bitmap b - (Bitmap)rm.GetObject("happyDude");
        p.Image - (Image)b;
        p.Height = b.Height;
        p.Width - b.Width;
        p.Location » new Point(10, 10);

        // Загружаем ресурс-текст
        Label label1 - new Label();
        label1.Location = new Point(50, 10);
        label1.Font - new Font(label1.Font.FontFamily, 12, FontStyle.Bold);
        label1.AutoSize = true;
        label1.Text = rm.GetString("welcomeString");

        // Создаем форму и настраиваем ее
        Form f = new Form();
        f.Height - 100;
        f.Width - 370;
        f.Text - "These resources are embedded in the assembly!";

        // Добавляем элементы управления на форму
        f.Controls.Add(p);
        f.Controls.Add(label1);
        f.ShowDialog();
    }
}
```

Перед запуском приложения не забудем изменить метод Main(), поместив туда вызов метода ReadMyResources:

```
staic void Main(string[] args)
{
    ...
    MyResourceReader r = new MyResourceReaderO;
    r.ReadMyResources();
}
```

То, что должно получиться, представлено на рис. 9.34.

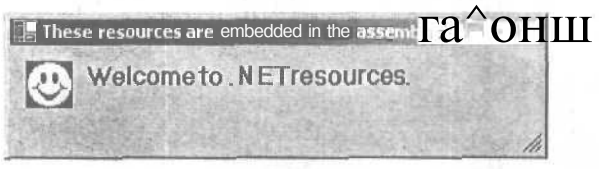


Рис. 9.34. Извлечение ресурсов при помощи ResourceManager

Код приложения ResourceTest можно найти в подкаталоге Chapter 9.

Работа с ресурсами при помощи IDE Visual Studio.NET

Конечно же, проще всего добавлять ресурсы в сборку при помощи возможностей, предоставляемых в наше распоряжение Visual Studio.

При создании нового приложения с использованием шаблона Windows Application Visual Studio автоматически создаст файл *.resx. При добавлении в этот проект новых ресурсов в файл *.resx будут автоматически добавляться новые пары имя — значение. Увидеть файл *.resx можно при помощи кнопки Show all files (Показать все файлы) в окне Solution Explorer (рис. 9.35).

Настроить параметры для этого файла можно, выбрав его в окне Solution Explorer и нажав кнопку Properties (Свойства). По умолчанию для свойства Build Action (Действие при создании), которое будет применяться к данному файлу ресурсов, установлено значение Embedded Resource (Встроенный ресурс) — см. рис. 9.36. Это значит, что данный файл ресурсов будет вложен в сборку.

Рис. 9.35. Как увидеть файл *.resx

Рис. 9.36. Настройка свойства Build Action для файла *.resx

Давайте поработаем со вложением ресурсов при помощи Visual Studio на примере. Мы создадим новый проект C# на основе шаблона Windows Application и назовем его ResLoader. Главная форма этого приложения содержит два объекта PictureBox: первый будет содержать изображение из файла happy.bmp (при помощи свойства Image), а второй будет пустым. Кроме того, на приложении будет размещена единственная кнопка, при нажатии на которую изображение будет динамически считываться из файла и помещаться в пустой PictureBox. Интерфейс нашего приложения представлен на рис. 9.37.

Рис. 9.37. Перед загрузкой изображения

При вставке в проект ресурсов (в нашем случае — изображения) Visual Studio автоматически создает объект класса ResourceManager в области видимости метода InitializeComponent():

```
private void InitializeComponent()
{
        System.Resources.ResourceManager resources = new System.Resources.ResourceManager
                                                        (typeof(MainForm));
        ...

        pictureBox1.Image = (System.Drawing.Image) resources.GetObject
                                                ("pictureBox1.Image");
}
```

Естественно, мы можем использовать другой объект ResourceManager для того же самого ресурса где угодно в другом месте нашего приложения. Например, вот код для события Click нашей единственной кнопки:

```
// Проверьте, что у вас присутствует строка "using System.Resources"
private void btnLoadRes_Click(object sender, System.EventArgs e)
{
        // Создаем объект ResourceManager
        ResourceManager resources = new ResourceManager (typeof(MainForm));

        // Считываем изображение из хранилища ресурсов сборки и помещаем его во второй
        // PictureBox
        this.pictureBox2.Image = ((System.Drawing.Bitmap)
                    (resources.GetObject("pictureBox1.Image")));

        // Все сделано!
        resources.ReleaseAllResources();
}
```

При нажатии на кнопку в приложении мы увидим, что изображение будет извлечено из внутреннего хранилища ресурсов сборки и помещено во второй PictureBox (рис. 9.38).

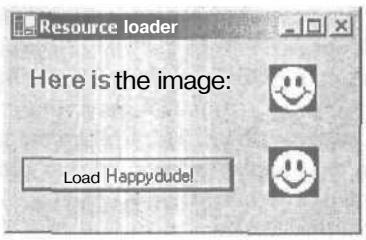


Рис. 9.38. После загрузки изображения

Код приложения ResLoader можно найти в подкаталоге Chapter 9.

Подведение итогов

Возможности GDI+ реализуются при помощи многочисленных типов из пространства имен .NET. Глава начинается с рассмотрения типов из пространства имен System.Drawing и анализа общих принципов работы с событиями Paint. Главное средство для вывода графики в GDI+ — это, без сомнения, класс Graphics.

Большая часть этой главы посвящена особенностям работы с различными типами для вывода графических объектов. Класс Pen (перо) позволяет выводить изображения при помощи линий, класс Brush (кисть) — закрашивать внутреннее пространство геометрических фигур. При помощи типов из пространства имен System.Drawing.Drawing2D мы можем обеспечить дополнительные очень интересные возможности работы с перьями и кистями, например градиентные или текстурные заливки. Применение типа Font требует большого объема дополнительной информации (об используемых объектоах Brush и Color, местонахождения выводимого текста и т. п.), однако обеспечивает множество возможностей.

Глава завершается рассмотрением форматов ресурсов .NET и возможностей работы с ними. Как мы могли убедиться, ресурсы (изображения, текстовые строки и т. п.) совершенно не обязательно встраивать в сборки, хотя размещение в сборках обеспечивает их защиту и удобство при развертывании приложения. Существует несколько форматов файлов ресурсов .NET. Часто работа начинается с создания файлов *.resx в формате XML. Далее при помощи утилиты resgen.exe эти файлы можно преобразовать в двоичный формат — *.resources. Затем эти двоичные файлы можно встроить в сборку. Кроме того, при использовании IDE Visual Studio.NET можно и не задумываться об этих операциях — внедрение ресурсов в сборку будет произведено автоматически. Для доступа к ресурсам в процессе выполнения обычно используется класс ResourceManager.

Элементы управления 10

Эту главу можно воспринимать как путеводитель по многочисленным элементам управления, используемым в приложениях с графическим интерфейсом и определенным в пространстве имен System.Windows.Forms. В главе 8 мы уже рассказывали о некоторых элементах управления уровня формы, таких как MainMenu (главное меню), MenuItem (элемент меню), StatusBar (строка состояния) и ToolBar (панель инструментов). В этой главе мы рассмотрим элементы управления, которые обычно размещаются в клиентской части формы (кнопки, текстовые поля, панели и т. п.).

Помимо этого, мы рассмотрим также дополнительные темы, тесно связанные с элементами управления, например, порядок перехода между элементами управления, «закрепление» элемента управления в определенном месте формы и т. п.

Глава заканчивается обсуждением моментов, связанных с созданием собственных диалоговых окон, включая средства обработки данных, вводимых пользователем, и проверки допустимости введенных значений. Кроме того, мы познакомимся с новой возможностью архитектуры форм .NET: наследованием между формами.

Иерархия классов элементов управления

Классы, представляющие графические элементы управления, находятся в пространстве имен System.Windows.Forms. С их помощью обеспечивается реакция на действия пользователя в приложении Windows Forms. Классы элементов управления связаны между собой достаточно сложными отношениями наследования. Общая схема таких отношений представлена на рис. 10.1 (обратите внимание, что все они происходят от единственного общего класса Control).

Как уже говорилось в главе 8, класс Control как общий предок обеспечивает все производные классы общим набором важнейших возможностей. В числе этих возможностей можно перечислить события мыши и клавиатуры, физические разме-

ры и местонахождение элемента управления (свойства Height, Width, Left, Right, Location), установку цвета фона и цвета переднего плана, выбор шрифта и т. п. В этой главе мы будем считать, что с возможностями, унаследованными от класса Control, читатель уже знаком (по главе 8), и мы уделим основное внимание уникальным членам каждого элемента управления.

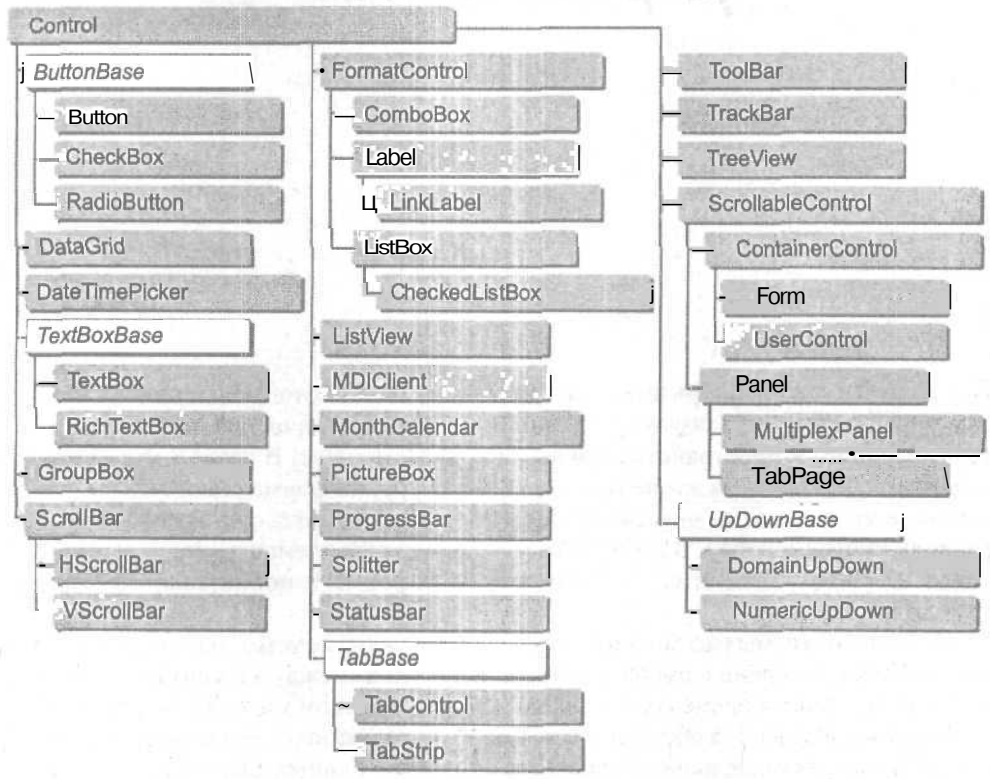


Рис. 10.1. Иерархия элементов управления Windows Forms

Как вручную добавить элементы управления на форму

Вне зависимости от того, какой именно элемент управления добавляется на форму, для его размещения нам придется выполнять один и тот же набор операций. Прежде всего нам нужно создать объект соответствующего класса. Следующая задача — при помощи свойств, методов и событий данного объекта настроить его параметры (обычно это делается либо в конструкторе формы, либо в методе InitializeComponents(). И наконец, последнее, что обязательно необходимо сделать, — добавить элемент управления в коллекцию Controls для данной формы. Если мы пропустим этот последний этап, то элемент управления просто не будет видим на форме! Все три этапа можно проиллюстрировать на примере:

```
// Не забудьте добавить ссылку на System.Windows.Forms.dll
using System.Windows.Forms;

class MyForm : Form
{
        // 1) Создаем объект элемент управления
        private TextBox firstNameBox = new TextBox();

        MyForm()
        {
                this.Text = "Controls in the raw";

                // 2) Настраиваем созданный TextBox
                firstNameBox.Text = "Chucky";
                firstNameBox.Size = new Size(150, 50);
                firstNameBox.Location = new Point(10, 10);

                // 3) Добавляем TextBox в коллекцию Controls
                this.Controls.Add(firstNameBox);
        }
...
}
```

Класс Control$ControlCollection

Несмотря на то что процесс добавления на форму нового элемента управления обычно не вызывает вопросов, все же свойство Controls формы заслуживает более подробного рассмотрения. Это свойство возвращает ссылку на вложенный класс ControlCollection, определенный внутри класса Control (то есть его полное имя будет Control$ControlCollection). Внутри ControlCollection создается запись для каждого из элементов управления, размещенных на форме. Мы можем получить ссылку на этот объект в любой момент, когда в процессе выполнения программы нам понадобится обратиться к списку элементов управления формы:

```
// Получаем ссылку на объект Control$ControlCollection для формы
Control.ControlCollection coll = this.Controls;
```

После получения ссылки в нашем распоряжении — любые члены ControlCollection. Их перечень представлен в табл. 10.1.

Таблица 10.1. Члены вложенного класса ControlCollection

Члены	Назначение
Add() AddRange()	Используется для добавления на форму нового элемента управления (массива элементов управления)
Clear()	Удаляет все элементы из коллекции
Count	Возвращает количество элементов в коллекции
GetChildIndex() SetChildIndex()	Позволяет установить или получить номер указанного пользователем элемента в коллекции
GetEnumerator()	Возвращает интерфейс IEnumerator для коллекции
Remove()	Удаляет указанный пользователем элемент управления из коллекции (указывается номер этого элемента)

Проиллюстрируем возможности ControlCollection на примере. Мы добавим на форму еще один элемент управления — кнопку, и в обработчике события Click этой кнопки мы будем выводить информацию о каждом из членов коллекции:

```
class MyForm : Form
{
        private TextBox firstNameBox = new TextBox();
        private Button btnShowControls = new Button();

        MyForm()
        {
                // Настраиваем TextBox
                ...

                // Добавляем новую кнопку
                btnShowControls.Text = "Examine Controls collection";
                btnShowControls.Size - new Size(90, 90);
                btnShowControls.Location = new Point(10, 70);
                btnShowControls.Click += new EventHandler(btnShowControls_Clicked);
                this.Controls.Add(btnShowControls);
        }

        protected void btnShowControls_Clicked(object sender, EventArgs e)
        {
                // Выводим информацию о каждом элементе в коллекции
                Control.ControlCollection coll = this.Controls;
                foreach(Control c in coll)
                {
                        // Второй параметр GetChildIndex() определяет, будет ли
                        // сгенерировано исключение, если эленент не обнаружен
                        if(c != null)
                                MessageBox.Show(c.Text, "Index numb: " +
                                                coll.GetChildIndex(c, false));
                }
        }
...
}
```

Графический интерфейс нашего приложения представлен на рис. 10.2. При нажатии на кнопку откроются два окна сообщения с информацией об элементах управления формы (рис. 10.3).

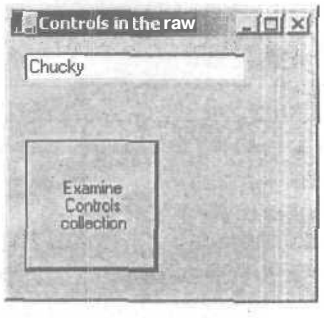


Рис. 10,2. Элементы управления на форме

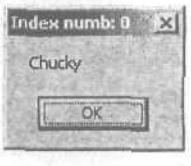


Рис. 10.3. Вывод информации об этих элементах управления

Код приложения ControlsByHand можно найти в подкаталоге Chapter 10.

Как добавить элементы управления на форму при помощи Visual Studio

Скорее всего, при создании реального приложения мы будем добавлять элементы управления на форму не вручную, а при помощи графических средств Visual Studio. Обычно достаточно выбрать нужный элемент управления в Tool 8ox и поместить его на форме. Visual Studio автоматически сгенерирует нужный код для нашей формы. После этого можно изменить название элемента управления на более содержательное (например, вместо buttonl, предлагаемого по умолчанию, — bntFirstName).

Visual Studio, конечно же, позволяет не только размещать на форме элементы управления, но и настраивать их свойства. Для этого достаточно щелкнуть на элементе управления правой кнопкой мыши и в контекстном меню выбрать Properties (Свойства). Все изменения, которые мы произведем в открывшемся окне (рис. 10.4), будут добавлены в код метода InitializeComponents().

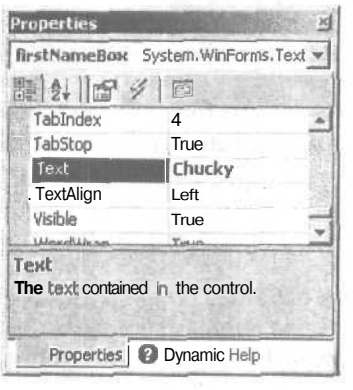


Рис. 10.4. Настройка элементов управления средствами Visual Studio

То же самое окно позволяет настроить не только свойства данного элемента управления, но и обработку событий этого элемента. Перейти в список событий можно при помощи кнопки, показанной на рис. 10.5. Выберем в списке нужное нам событие и рядом с ним введем имя метода (или выберем метод из списка).

Если мы настроим свойства firstNameBox так, как это представлено на иллюстрациях, то обнаружим в определении метода InitializeComponent() следующий код (обратите внимание, что элемент управления помещается в коллекцию ControlCollection при помощи метода AddRange()):

```
private void InitializeComponents()
{
        this.firstNameBox = new System.Windows.Forms.TextBox();
        this.firstNameBox.Location = new System.Drawing.Point(32, 40);
        this.firstNameBox.TabIndex = 0;
        this.firstNameBox.Text = "Chucky";
        this.firstNameBox.TextChanged += new
                    System.EventHandler(this.firstNameBox_TextChanged);
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
```

```
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Controls.AddRange(new System.Windows.Forms.Control[] {this.FirstNameBox});
        ...
    }
```

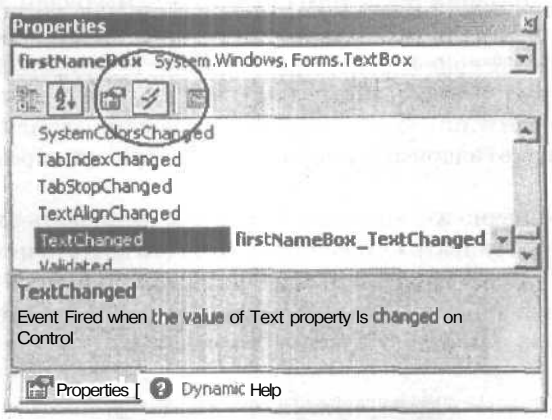


Рис. 10.5. Настройка обработчиков событий

Кроме того, в наше распоряжение будет предоставлена «заготовка» для метода firstNameBox_TextChanged (если мы введем это имя в окне событий для firstNameBox):

```
protected void firstNameBox_TextChanged (object sender, System.EventArgs e)
{
    // Все, что вам нужно от этого метода...
}
```

В дальнейшем мы будем работать со свойствами и методами элементов управления только вручную — для лучшего понимания. Помните, что все то же самое можно сделать (возможно, меньшими усилиями) при помощи IDE Visual Studio.

Элемент управления TextBox

Элемент управления TextBox (текстовое окно) предназначен для хранения текста (одной или нескольких строк). При желании текст в TextBox может быть настроен как «только для чтения», а в правой и нижней части можно поместить полосы прокрутки. Класс TextBox происходит непосредственно от класса TextBoxBase. TextBoxBase обеспечивает общими возможностями как TextBox, так и RichTextBox. Свойства, определенные в TextBoxBase, представлены в табл. 10,2.

Таблица 10.2. Свойства TextBoxBase

Свойство	Назначение
AcceptsTab	Определяет, что будет производиться при нажатии на клавишу Tab: вставка символа табуляции в само поле или переход к другому элементу управления
AutoSize	Определяет, будет ли элемент управления автоматически изменять размер при изменении шрифта на нем

Свойство	Назначение
BackColor ForeColor	Позволяют получить или установить значение цвета фона и переднего плана
HideSelection	Позволяет получить или установить значение, определяющее, будет ли текст в TextBox оставаться выделенным после того, как этот элемент управления будет выведен из фокуса
MaxLength	Определяет максимальное количество символов, которое можно будет ввести в TextBox
Modified	Позволяет получить или установить значение, определяющее, был ли текст в TextBox изменен пользователем
Multiline	Указывает, может ли TextBox содержать несколько строк текста
Readonly	Помечает TextBox как «только для чтения»
SelectedText SelectionLength	Содержат выделенный текст (или определенное количество символов) в TextBox
SelectionStart	Позволяет получить начало выделенного текста в TextBox
Wordwrap	Определяет, будет ли текст в TextBox автоматически переноситься на новую строку при достижении предельной длины строки

В TextBoxBase также определено множество методов: для работы с буфером обмена (Cut(), CopyC) и Paste(J), отменой ввода (Undo()) и прочими возможностями редактирования (Clear O, AppendText O и т. п.).

Из всех событий, определенных в TextBoxBase, наибольший интерес представляет событие TextChange (его примерным аналогом можно считать сообщение EN_CHANGE в Win32). Это событие происходит при изменении текста в объекте класса, производном от TextBoxBase. Например, его можно использовать для проверки допустимости вводимых пользователем символов (например, предположим, что пользователь должен вводить в поле только цифры или, наоборот, только буквы).

Свойства, унаследованные от Control и от TextBoxBase, определяют большую часть возможностей TextBox. Свойств, определенных непосредственно в классе TextBox, не так уж и много. Они представлены в табл. 10.3.

Таблица 10.3. Свойства, определенные в классе TextBox

Свойство	Назначение
AcceptsReturn	Позволяет определить, что происходит, когда пользователь при вводе текста нажал на Enter. Варианта два: либо в TextBox начинается новая строка текста, либо активизируется кнопка по умолчанию на форме
CharacterCasing	Позволяет получить или установить значение, определяющее, будет ли изменяться регистр вводимых пользователем символов
PasswordChar	Позволяет выбрать символ, используемый для отображения вводимых пользователем данных (в поле для ввода пароля)
ScrollBars	Позволяет получить или установить значение, определяющее, будут ли в TextBox с несколькими строками присутствовать полосы прокрутки
TextAlign	Позволяет определить выравнивание текста в TextBox (используются значения из перечисления HorizontalAlignment)

Значения перечисления HorizontalAlignment представлены в табл. 10.4.

Таблица 10.4. Значения перечисления HorizontalAlignment

Значение	Описание
Center	Выравнивание по центру
Left	Выравнивание по левому краю
Right	Выравнивание по правому краю

Некоторые возможности TextBox

Чтобы проиллюстрировать возможности TextBox, мы создадим форму с этим элементом управления. При этом текстовое окно будет многострочным, оно будет реагировать на Enter и Tab и в нем будет вертикальная полоса прокрутки. Код для настройки объекта TextBox (будем считать, что этот объект уже создан и называется multiLineBox) будет выглядеть так:

```
multiLineBox.Location = new System.Drawing.Point (152. 8);
multiLineBox.Text = "Type some stuff here (and hit the return and tab keys...)";
multiLineBox.Multiline = true;
multiLineBox.AcceptsReturn - true;
multiLineBox.ScrollBars - System.Windows.Forms.ScrollBars.Vertical;
multiLineBox.TabIndex * 0;
multiLineBox.AcceptsTab = true;
```

Обратите внимание, что для настройки полос прокрутки мы настраиваем свойство ScrollBars. Для него используются значения из перечисления ScrollBars (возможные варианты — Vertical, Horizontal, None, Both).

Для того чтобы проиллюстрировать работу со свойствами TextBox непосредственно во время выполнения программы, мы создадим на форме кнопку. Обработчик события Click этой формы выведет текст из TextBox в окне сообщения. Пользовательский интерфейс нашего приложения представлен на рис. 10.6.

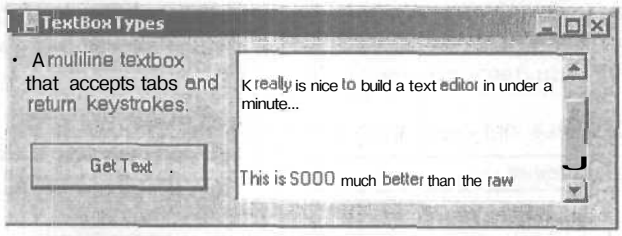


Рис. 10.6. Работаем с текстом в TextBox во время выполнения программы

Код обработчика события Click для кнопки будет очень простым:

```
protected void btnGetMultiLineText_click (object sender, System.EventArgs e)
{
        MessageBox.Show(multiLineBox.Text, "Here is your text");
}
```

То, что должно получиться при нажатии на кнопку, представлено на рис. 10.7.

В следующем примере (рис. 10.8) мы добавим на форму еще несколько текстовых окон — на этот раз чтобы проиллюстрировать возможности отображения текста. Второе текстовое окно (capsOnlyBox) будет отображать любой вводимый текст

исключительно заглавными буквами, а третье (passwordBox) будет скрывать вводимый текст за специальными символами (в нашем случае — за символом $). А еще мы разместим рядом с passwordBox специальную кнопку (btnPasswordDecoderRing), по которой вводимый в «парольном» текстовом окне текст будет расшифровываться:

```
protected void btnGetMultiLineText_Click (object sender, System.EventArgs e)
{
      MessageBox.Show(multiLineBox.Text, "Here is your text");
}
```

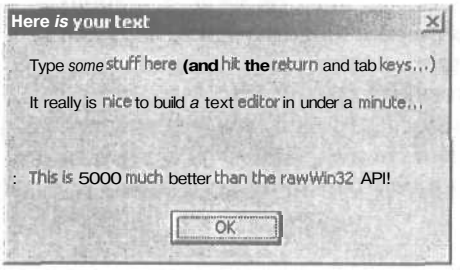


Рис. 10.7. Извлекаем введенные пользователем в TextBox данные

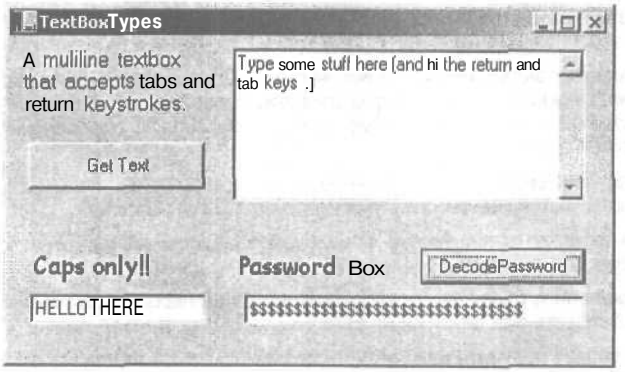


Рис. 10.8. Возможности отображения текста в текстовом окне

Код для новых текстовых окон может выглядеть следующим образом:

```
// Для поля, в котором символы автоматически преобразуются в заглавные, используется
// свойство CharacterCasing. Допустимые для него значения (из соответствующего
// перечисления) - Upper (только заглавные), Lower Столько строчные) и Normal
// (по умолчанию, изменений не будет).
capsOnlyBox.Location = new System.Drawing.Point (14. 176);
capsOnlyBox.CharacterCasting = System.Windows.Forms.CharacterCasing.Upper;
capsOnlyBox.Size = new System.Drawing.Size (120, 20):

// "Парольное" текстовое окно
passwordBox.Location = new System.Drawing.Point (160. 176):
passwordBox.PasswordChar = '$';
```

Как уже говорилось, у TextBoxBase существует еще один производный класс - RichTextBox. Этот элемент управления представляет текстовое окно для ввода и отображения текста с форматированием. Например, мы можем настроить выбор

шрифта для участков текста, выделение гиперссылок, создание маркированных и нумерованных списков и т. п.

Код приложения TextBoxes можно найти в подкаталоге Chapter 10.

Великая и могучая кнопка (а также родительский класс ButtonBase)

Кнопка (button) — это самый простой из всех элементов управления и при этом наиболее часто используемый. Можно сказать, что кнопка — это возможность принять ввод (щелчок кнопкой мыши или набор на клавиатуре) наиболее простым способом. Непосредственный предок класса System.Windows.Form.Button в иерархии классов .NET — это класс ButtonBase, обеспечивающий общие возможности для целой группы производных от него элементов управления (таких как Button, CheckBox и RadioButton). Некоторые (конечно же, далеко не все) свойства ButtonBase представлены в табл. 10.5.

Таблица 10.5. Свойства ButtonBase

Свойство	Назначение
FlatStyle	Позволяет настроить «рельефность» кнопки. Используются значения из перечисления FlatStyle
Image	Позволяет задать изображение, которое будет выводиться на кнопке (при этом можно указать точное местонахождение изображения). Фоновый рисунок лучше настраивать при помощи свойства BackgroundImage, определенного в базовом классе Control
ImageAlign	Позволяет определить выравнивание изображения, размещенного на кнопке. Используются значения из перечисления ContentAlignment
ImageIndex ImageList	Эти свойства используются для работы с набором изображений (объектом ImageList), выводимых на кнопке
IsDefault	Определяет, будет ли эта кнопка являться кнопкой по умолчанию (то есть срабатывать при нажатии на Enter)
TextAlign	Позволяет получить или установить выравнивание текста на кнопке. Также используются значения из перечисления ContentAlignment

Чуть-чуть подробнее о свойстве FlatStyle. Это свойство определяет внешний вид кнопки (точнее, ее «рельефность», «объемность») и для него используются значения из перечисления FlatStyle (табл. 10,6).

Таблица 10.6. Значения перечисления FlatStyle

Значение	Описание
Flat	Кнопка будет выглядеть плоской, безо всякого намека на трехмерность. При прохождении над ней указателя мыши ее цвет будет меняться
Popup	Кнопка будет выглядеть плоской, однако при появлении над ней указателя мыши она превратится в «объемную
Standard	Кнопка будет выглядеть трехмерной, как большинство кнопок в известных вам приложениях
System	Вид кнопки будет определяться настройками операционной системы пользователя

Сам класс Button не определяет каких-либо дополнительных возможностей помимо унаследованных от ButtonBase, за единственным, но существенным исключением свойства DialogResult. Как мы увидим далее, это свойство позволяет возвращать значение при закрытии диалогового окна, например, при нажатии кнопок OK или Cancel (Отменить).

Выравнивание текста и изображений относительно краев кнопки

В подавляющем большинстве случаев выравнивание текста, размещенного на кнопке, производится по центру, так что текст будет размещен строго посередине кнопки. Однако если нам по каким-то причинам необходимо использовать другой стиль выравнивания, в нашем распоряжении — свойство TextAlign, определенное в классе ButtonBase. Для TextAlign используются значения из перечисления ContentAlignment (табл. 10.7). Значения из того же перечисления используются и для определения положения изображения на кнопке (чем мы еще займемся).

Таблица 10.7. Значения перечисления ContentAlignment

Значение	Описание (выравнивание)
BottomCenter	По нижнему краю кнопки, относительно боковых краев — посередине
BottomLeft	Но нижнему краю кнопки, слева
BottomRight	По нижнему краю кнопки, справа
MiddleCenter	По центру кнопки
MiddleLeft	Относительно верхнего и нижнего краев — по центру, относительно боковых краев — слева
MiddleRight	Относительно верхнего и нижнего краев — по центру, относительно боковых краев — справа
TopCenter	По верхнему краю кнопки, относительно боковых краев — посередине
TopLeft	По верхнему краю кнопки, слева
TopRight	По верхнему краю кнопки, справа

Некоторые возможности работы с кнопками

Проиллюстрируем возможности работы с кнопками на примере. В нашем приложении будут использованы свойства FlatStyle, ImageAlign и TextAlign. При этом при возникновении события Click для одной из кнопок нашего приложения (эта кнопка будет расположена по центру формы) будут изменяться как находящийся на ней текст, так и его выравнивание относительно краев. При этом текст будет отображать текущее значение свойства TextAlign, а его размещение явится прямым результатом применения этого значения. Для интереса мы заставим одновременно меняться местонахождение изображения на еще одной кнопке (bindImage), на которой, помимо всего прочего, мы также разместим фоновый рисунок. То, что должно получиться, представлено на рис. 10.9.

Код нашего приложения может быть таким:

```
public class ButtonForm: System.Windows.Forms.Form
{
```

```
        // На форме будет четыре кнопки
        private System.Windows.Forms.Button btnImage;
        private System.Windows.Forms.Button btnStandard;
        private System.Windows.Forms.Button btnPopup;
        private System.Windows.Forms.Button btnFlat;

        // Переменная для хранения текущего значения выравнивания
        ContentAlignment currAlignment = ContentAlignment.MiddleCenter;
        int currEnumPos = 0;

        // Код InitializeComponent() мы приводить не будем...
        protected void btnStandard_Click (object sender, System.EventArgs e)
        {
            // Получаем все варианты допустимых значений из перечисления
            CurrentAlignment
            Array values = Enum.GetValues(currAlignment.GetType());

            // Увеличиваем текущее значение currEnumPos на 1. Если достигнуто
            // последнее значение перечисления, возвращаемся к первому
            currEnumPos++;
            if(currEnumPos >= values.Length)
                CurrEnumPos = 0;

            // Меняем текущее значение перечисления
            currAlignment =
                (ContentAlignment)ContentAlignment.Parse(currAlignment.GetType(),
                                        values.GetValue(currEnumPos).ToString());

            // Выводим значение на кнопке и устанавливаем выравнивание
            btnStandard.Text = currAlignment.ToString();
            btnStandard.TextAlign = currAlignment;

            // А теперь устанавливаем выравнивание для изображения
            // на кнопке btnImage
            btnImage.ImageAlign = currAlignment;
        }
        ...
}
```

Код приложения Buttons можно найти в подкаталоге Chapter 10.

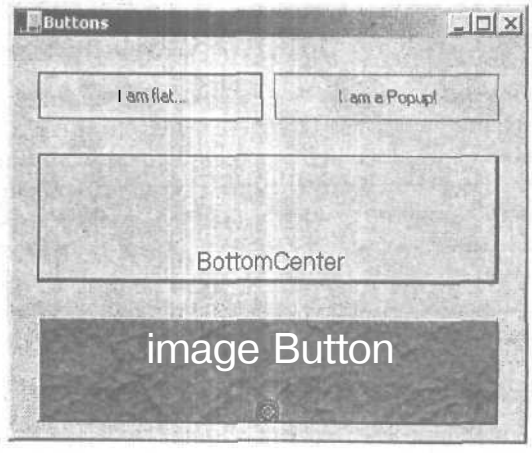


Рис. 10.9. Возможности кнопок в действии

Работаем с флажками

Будем считать, что размещать на форме кнопки и настраивать их параметры мы уже умеем. Следующее, с чем мы должны познакомиться — это флажки (тип CheckBox), для которых в .NET предусмотрено три возможных состояния. Как и тип Button, CheckBox наследует большую часть своих возможностей от базовых классов Control и ButtonBase. Однако в этом классе существуют и свои собственные члены, обеспечивающие дополнительные уникальные возможности. Наиболее важные свойства CheckBox представлены в табл. 10.8.

Таблица 10.8. Свойства класса CheckBox

Свойство	Назначение
Appearance	Настраивает вид флажка. Для этого свойства используются значения из перечисления Appearance
AutoCheck	Позволяет получить или установить значение, определяющее, будут ли значения Checked и CheckState, а также внешний вид флажка автоматически изменяться при щелчке на нем
CheckAlign	Позволяет установить горизонтальное и вертикальное выравнивание собственно флажка (квадратика) в элементе управления CheckBox. Используются значения из перечисления ContentAlignment
Checked	Возвращает значение типа bool, представляющее текущее состояние флажка (выбран или не выбран). Если для свойства ThreeState установлено значение true, то свойство Checked будет возвращать true как для явно выбранного флажка, так и для того флажка, для которого установлено значение «не определено» (indeterminate)
CheckState	Позволяет получить или установить значение флажка (установлен — не установлен — не определено), используя не true и false, как в Checked, а три значения из перечисления CheckState. Обычно используется, если свойство ThreeState для флажка имеет значение true (то есть он допускает три значения)
ThreeState	Определяет, будут ли для флажка использоваться три значения (из перечисления CheckState) или только два

Возможные состояния флажка (Indeterminate можно использовать только тогда, когда для свойства ThreeState установлено значение true) представлены в табл. 10.9.

Таблица 10.9. Значения перечисления CheckState

Значение	Описание
Checked	Флажок установлен
Indeterminate	Значение не определено (обычно флажок выглядит как «серый», затененный)
Unchecked	Флажок снят

Вы наверняка видели флажки во всех трех состояниях. Состояние «значение не определено» (indeterminate) может быть установлено, например, для верхнего элемента иерархии, в которой для одной части подчиненных элементов флажок установлен, а для другой — снят.

Работаем с переключателями и группирующими рамками

Скорее всего, тип RadioButton (переключатель) не требует подробных пояснений. Этот тип можно воспринимать как несколько видоизмененный флажок, при этом сходство между этими типами подчеркивается почти полным совпадением наборов членов. Между типами RadioButton и CheckBox существуют лишь два важных различия: в RadioButton предусмотрено событие CheckedChanged (возникающее при изменении значения Checked), а кроме того, RadioButton не поддерживает свойство ThreeState и не может принимать состояние Indeterminate (не определено).

Переключатели всегда используются в группах, которые рассматриваются как некое единое целое. Внутри группы переключателей одновременно может быть выбран только один переключатель. Например, логично будет объединить в группу четыре объекта RadioButton для того, чтобы предоставить пользователю возможность выбрать цвет автомобиля: вряд ли будет удобно, если пользователь выберет одновременно два цвета. Для группировки переключателей в группы используется тип GroupBox. Едва ли есть необходимость рассматривать его члены: практически все его возможности обеспечиваются базовым классом Control.

Некоторые возможности переключателей и флажков

Проиллюстрируем возможности типов CheckBox, RadioButton и GroupBox на примере нового приложения, которое будет называться CarConfig. На главной форме будет определено несколько элементов управления, при помощи которых пользователь выберет некоторые характеристики автомобиля. Интерфейс нашего приложения представлен на рис. 10.10.

Рис. 10.10. Флажок и переключатели в группирующей рамке на форме

Код, относящийся к созданию элементов управления, мы приводить не будем — начнем сразу с настройки их параметров. Прежде всего мы настроим свойства флажка:

```
// Настраиваем свойства флажка
checkFloorMats.Location = new System.Drawing.Point (16, 16):
checkFloorMats.Text = "Extra Floor Mats";
checkFloorMats.Size = new System.Drawing.Size (136. 24):
checkFloorMats.FlatStyle = System.Windows.Forms.FlatStyle.Popup;

// Добавляем его в коллекцию Controls
this.Controls.Add(this.checkFloorMats);
```

С точки зрения программного кода добавление элементов управления (переключателей) в группирующую рамку — это добавление их объектов в коллекцию Control s соответствующего объекта GroupBox. Производится оно точно так же, как и добавление элементов управления в коллекцию Controls для формы. Чтобы было интереснее, мы еще и настроим обработчики событий Enter и Leave для нашего объекта GroupBox. Эти события возникают, когда фокус переходит к объекту в группирующей рамке и к объекту за его пределами:

```
// Настраиваем "желтый" переключатель
radioYellow.Location - new System.Drawing.Point (96, 24);
radioYellow.Text = "Yellow";
radioYellow.Size - new System.Drawing.Size (64, 23);

// "Зеленый", "красный" и "розовый" переключатели настраиваются
// аналогичным образом...

// А теперь создаем группирующую ранку и помещаем в нее переключатели
groupBox1.Location = new System.Drawing.Point (16. 56);
groupBox1.Text - "Exterior Color";
groupBox1.Size = new System.Drawing.Size (264. 88);

groupBox1.Leave +- new System.EventHandler (groupBox1_Leave);
groupBox1.Enter += new System.EventHandler (groupBox1_Enter);

groupBox1.Controls.Add (this.radioPink);
groupBox1.Controls.Add (this.radioYellow);
groupBox1.Controls.Add (this.radioRed);
groupBox1.Controls.Add (this.radioGreen);
```

В подавляющем большинстве случаев обработчики событий Enter и Leave для группирующей рамки не используются. Мы (исключительно для примера) настроим их таким образом, что при возникновении этих событий текст в заголовке рамки будет изменяться:

```
// Реагируем на получение и потерю фокуса эленентаии в группирующей рамке
protected void group8oxl_Leave (object sender. System.EventArgs e)
{
        groupBoxl.Text = "Exterior Color: Thanks for visiting the group...";
}

protected void groupBoxl_Enter (object sender. System.EventArgs e)
{
        groupBoxl.Text - "Exterior Color: You are in the group...";
}
```

Последние элементы управления на форме — текстовое поле infoLabel и кнопку btnOrder также необходимо настроить. При щелчке на кнопке btnOrder текст в infoLabel будет изменяться:

```
protected void btnOrder_Click (object sender. System.EventArgs e)
{
        // Создаем переменную для хранения отображаемой информации
        string orderInfo = "";

        if(checkFloorMats.Checked)
                orderInfo += "You want floor mats.\n";

        if(radioRed.Checked)
                orderInfo += "You want a red exteriorAn";

        if(radioYellow.Checked)
                orderInfo +- "You want a yellow exterior.\n";

        if( radioGreen.Checked)
                orderInfo += "You want a green exterior.\n";

        if(radioPink.Checked)
                orderInfo += "Why do you want a PINK exterior?\n";

        // Выводим значение в текстовом поле
        infoLabel.Text = orderInfo;
}
```

И флажок (CheckBox), и переключатель (RadioButton) поддерживают свойство Checked, при помощи которого очень удобно получать информацию о состоянии соответственно флажка и переключателя. Однако если есть необходимость задействовать дополнительное третье состояние флажка (не определено — Indeterminate), то нам придется вместо Checked использовать свойство CheckState и значения из одноименного перечисления CheckState.

Элемент управления CheckedListBox

Типы Button, CheckBox и RadioButton являются производными от ButtonBase, и их можно определить как некие разновидности кнопок. Следующие несколько разделов этой главы будут посвящены типам, которые можно назвать членами семейства списков. К этим типам относятся CheckedListBox (список с флажками), ListBox (список) и ComboBox (комбинированный список). Первый на очереди — CheckedListBox.

Элемент управления CheckedListBox (список с флажками) позволяет помещать обычные флажки внутри поля с полосами прокрутки. Вот пример применения CheckedListBox: предположим, что в нашем приложении CarConfig мы с помощью этого элемента управления предоставили пользователям возможность выбирать дополнительные аудиоустройства (рис. 10.11).

Как и множество остальных элементов управления, подавляющее большинство возможностей CheckedListBox унаследовано от класса Control. Кроме того, некоторые члены CheckedListBox унаследованы от класса ListBox (от которого он производится напрямую). Класс ListBox будет рассмотрен чуть ниже.

Для добавления объектов в CheckedListBox используется метод Add() или AddRange() (для массива объектов). Объекты, передаваемые этим методам, должны быть типа string. Для каждого передаваемого значения создается флажок. Код для настройки CheckedListBox может выглядеть следующим образом:

```
// Настраиваем объект CheckedListBox
checkedListBoxRadioOptions.Location - new System.Drawing.Point (16. 48):
checkedListBoxRadioOptions.Cursor = Cursors.Hand;
checkedListBoxRadioOptions.Size = new System.Drawing.Size (256, 64);
checkedListBoxRadioOptions.CheckOnClick = true:

// Добавляем элементы в CheckedListBox
checkedBoxRadioOptions.Items.AddRange(new object[6] {"Front Speakers", "8-Track Tape
Player", "CD-Player". "Cassette Player". "Rear Speakers", "Ultra Base Thumper"});

// Как обычно, передаем созданный элемент управления а коллекцию Controls формы
this.Controls.Add (this.CheckedBoxradioOptions);
```

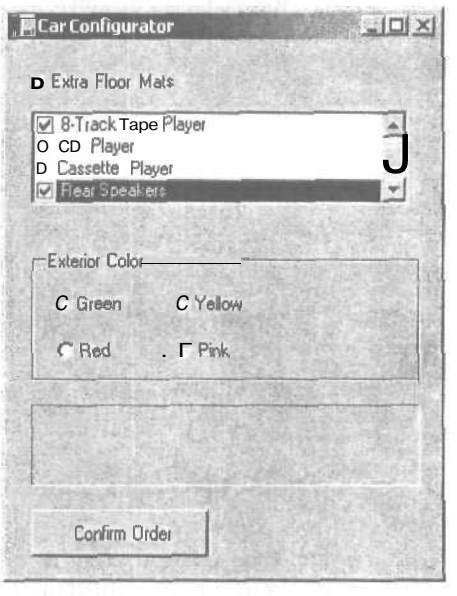


Рис. 10.11. Объект CheckedListBox на форме

Теперь мы должны обновить код для обработчика события Click кнопки btnOrder. Мы получаем информацию о том, какие элементы CheckedListBox выбраны в настоящее время, и добавляем их к значению переменной orderlnfo. Соответствующий код может выглядеть следующим образом:

```
protected void btnOrder_Click(object sender, System.EventArgs e)
{
        // Создаем переменную для хранения информации
        string orderlnfo = "";
        ...
        // Для каждого элемента в CheckedListBox:
        for(int i = 0: i < checkedBoxRadioOptions.Items.Count; i++)
        {
                // Выбран ли текущий элемент?
                if(checkedBoxRadioOptions.GetItemChecked(i))
                {
                        // Получаем текстовое значение для каждого элемента и добавляем его
                        // к orderlnfo
                        orderlnfo += "radio Item: ";
```

```
                orderInfo += checkedBoxRadioOptions.Items[i].ToString();
                orderInfo += "\n";
            }
        }
        ...
    }
```

Кроме того, в элементе управления CheckedListBox предусмотрена возможность использования нескольких столбцов. Для этого достаточно установить значение true для свойства MultiLine. Если мы внесем в наш код следующую строку:

```
checkedBoxRadioOptions.MultiColumn = true;
```

то наш список с флажками примет вид, представленный на рис. 10.12.

Рис. 10.12. CheckedListBox с несколькими столбцами

Списки

Как уже говорилось, CheckedListBox наследует большинство своих возможностей от типа ListBox. То же самое справедливо и в отношении класса ComboBox. Наиболее важные свойства System.Windows.Forms.ListBox представлены в табл. 10.10.

Помимо свойств в классе ListBox определены также многочисленные методы. Подавляющее большинство этих методов дублирует возможности, предоставляемые в наше распоряжение свойствами, поэтому мы их рассматривать здесь не будем.

Проиллюстрируем возможности ListBox на примере все того же нашего приложения. Мы добавим на форму объект ListBox, при помощи которого пользователь сможет выбрать марку автомобиля (BMW, Yugo и т. п.). Интерфейс приложения после добавления ListBox представлен на рис. 10.13.

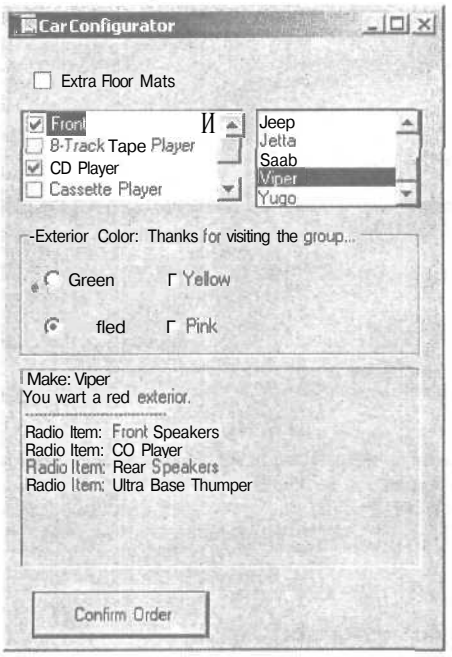


Рис. 10.13. Объект ListBox на форме

Таблица 10.10. Свойства класса ListBox

Свойство	Назначение
ScrollAlwaysVisible	Определяет, будет ли полоса прокрутки выводиться всегда
SelectedIndex	Индекс выделенного в настоящий момент элемента в списке (если такой имеется). Если ни один элемент не выделен, то возвращается значение -1
SelectedIndices	Набор индексов выделенных в настоящий момент элементов в списке. Если не выделен ни один элемент, то возвращается пустой набор
SelectedItem	Значение выделенного в настоящий момент элемента. Если ни один из элементов не выделен, то возвращается null
SelectedItems	Возвращает коллекцию значений выделенных элементов (для списков, в которых допускается выбор нескольких значений)
SelectionMode	Определяет число элементов, которые возможно выбрать в списке одновременно. Для этого свойства используются значения из перечисления SelectionMode
Sorted	Определяет, будут ли элементы в списке упорядочены (по алфавиту) или нет
TopIndex	Возвращает индекс первого видимого элемента в списке

Как обычно, первое, что мы должны сделать — создать объект класса ListBox (в нашем примере он будет называться carMakeList). Затем мы настроим параметры этого объекта и поместим его в коллекцию Controls формы:

```
// Настраиваем параметры списка
carMakeList.Location = new System.Drawing.Point (168, 48);
carMakeList.Size = new System.Drawing.Size (112. 67);
```

```
carMakeList.BorderStyle - System.Windows.Forms.BorderStyle.FixedSingle;
carMakeList.ScrollAlwaysVisible = true;
carMakeList.Sorted = true;

// Добавляем в список элементы при помощи метода AddRange()
carMakeList.Items.AddRange(new object[9] {"BMW", "Caravan", "Ford", "Grand Am", "Jeep",
"Jetta", "Saab", "Viper", "Yugo"});

// Добавляем новый элемент управления в коллекцию Controls формы:
this.Controls.Add(this.carMakeList);
```

Внесение изменений в обработчик события btnOrder_Click() также не составит труда:

```
protected void btnOrder_Click (object sender, System.EventArgs e)
{
        // Создаем переменную для хранения информации
        string orderInfo - "";
        ...
        // Получаем значение выбранного в настоящий момент элемента списка (обратите
        // внимание: значение, а не индекс)
if(carMakeList.SelectedItem !- null)
        orderInfo += "Make: " + carMakeList.SelectedItem + "\n";
        ...
}
```

Комбинированные списки

Как и списки (объекты ListBox), комбинированные списки (объекты ComboBox) позволяют пользователю производить выбор из списка заранее определенных элементов. Однако у комбинированных списков есть одно существенное отличие от обычных: пользователь может не только выбрать готовое значение из списка, но и ввести свое собственное. Класс ComboBox наследует большинство своих возможностей от класса ListBox (который, в свою очередь, является производным от Control), однако в нем предусмотрены и собственные важные свойства, представленные в табл. 10.11.

Таблица 10.11. Свойства ComboBox

Свойство	Назначение
DroppedDown	«Раскрывающийся вниз»: определяет, будет ли список ниспадающим
MaxDropDownItems	Определяет максимальное количество элементов, которое будет показано в нижней части ниспадающего списка. Допустимые значения — от 1 до 100
MaxLength	Определяет максимальную длину текста, который пользователь может ввести е ComboBox
SelectedIndex	Определяет индекс выделенного элемента ComboBox. Если ни один элемент не выделен, возвращается значение -1
SelectedItem	Возвращает ссылку на объект выделенного элемента ComboBox
SelectedText	Возвращает выделенный текст в поле редактирования ComboBox
SelectionLength	Определяет длину (в символах) выделенного текста в поле редактирования ComboBox
Style	Позволяет получить или установить стиль ComboBox. Для этого свойства используются значения из перечисления ComboBoxStyle
Text	Позволяет получить доступ к тексту в поле редактирования. При работе с ComboBox это унаследованное свойство используется чаще всех остальных

Стиль для ComboBox можно настроить при помощи свойства Style, для которого используются значения из перечисления ComboBoxStyle. В нашем распоряжении следующие варианты (табл. 10.12).

Таблица 10.12. Значения перечисления ComboBoxStyle

Значение	Описание
DropDown	Пользователь может вводить значения в поле редактирования. Для отображения списка пользователь должен нажать на кнопку со стрелкой, направленной вниз (Arrow Button)
DropDownList	Пользователь не может вводить значения в поле редактирования. Для отображения списка пользователь должен нажать на кнопку со стрелкой, направленной вниз (Arrow Button)
Simple	Пользователь может вводить значения в поле редактирования. Список значений виден всегда

Добавим элемент управления ComboBox на нашу форму. Он будет использоваться для выбора продавца, заполняющего форму заказа, и поэтому мы назовем его comboSalesPerson. Если продавец не найдет себя в списке, он сможет ввести свое имя в ComboBox самостоятельно. Новый вариант интерфейса пользователя представлен на рис. 10.14 (мы еще поменяли внешний вид объекта Label, придав ему более привлекательный вид).

Рис. 10.14. Новый вариант CarConfig — теперь с ComboBox

Как обычно, вначале мы создаем объект ComboBox, а затем настраиваем его параметры и помещаем в коллекцию Controls формы:

```
comboSalesPerson.Location = new System.Drawing.Point (152, 16)
comboSalesPerson.Size - new System.Drawing.Size (128, 21);
comboSalesPerson.Items.AddRange(new object[4] ("Baby Ry-Ry". "SPARK!". "Danny Boy".
"Karin 'Baby' Johnson"});
this.Controls.Add (this.comboSalesPerson);
```

Код изменений в btnOrder_Click() также несложен:

```
protected void btnOrder_Click (object sender. System.EventArgs e)
{
        // Создаем переменную для хранения информации
        string orderInfo = "";
        ...
        // Для получения данных о выбранном (или введенной пользователем) имени продавца
        // используем свойство Text
        if(comboSalesPerson.Text != "")
                orderInfo += "Sales Person: " + comboSalesPerson.Text +"\n";
        else
                orderInfo += "You did not select a sales person!" + "\n";
        ...
}
```

Настраиваем порядок перехода по Tab

Если на форме размещено несколько элементов управления, то пользователи обычно ожидают, что между ними можно будет перемещаться с помощью клавиши Tab. Часто бывает необходимо после размещения элементов управления настроить порядок перехода между ними. Для этого используются два свойства (унаследованные от базового класса Control и поэтому общие для всех элементов управления): TabStop и TabIndex.

Для свойства TabStop используются только два значения: true и false. Если для TabStop установлено значение true, то к этому элементу управления можно будет добраться с помощью клавиши Tab. Если же установлено значение false, то участвовать в переходах по Tab этот элемент управления не будет. Если элемент управления TabStop имеет значение true, то очередность перехода можно настроить с помощью свойства TabIndex:

```
// Настраиваем возможности перехода по Tab
radioRed.TabIndex = 2;
radioRed.TabStop = true;
```

Конечно же, эти свойства можно настроить и из окна свойств элемента управления (рис. 10.15).

Tab Order Wizard

В Visual Studio.NET предусмотрено средство, при помощи которого можно быстро настроить порядок перехода для элементов управления на форме. Это средство называется Tab Order Wizard и оно доступно из меню View (View ▸ Tab Order). Tab Order Wizard в действии представлен на рис. 10.16. Чтобы изменить значения TabIndex для каждого элемента управления, достаточно просто щелкать мышью на элементах управления в выбранном нами порядке перехода. Для элементов управления, помещенных в группирующую рамку, Tab Order Wizard создает отдельную последовательность перехода.

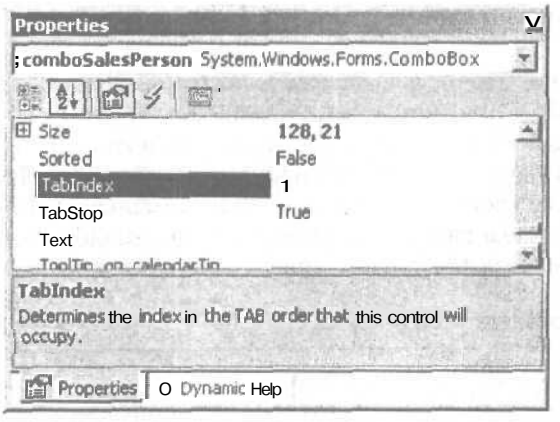


Рис. 10.15. Настройка TabIndex и TabStop из окна свойств элемента управления

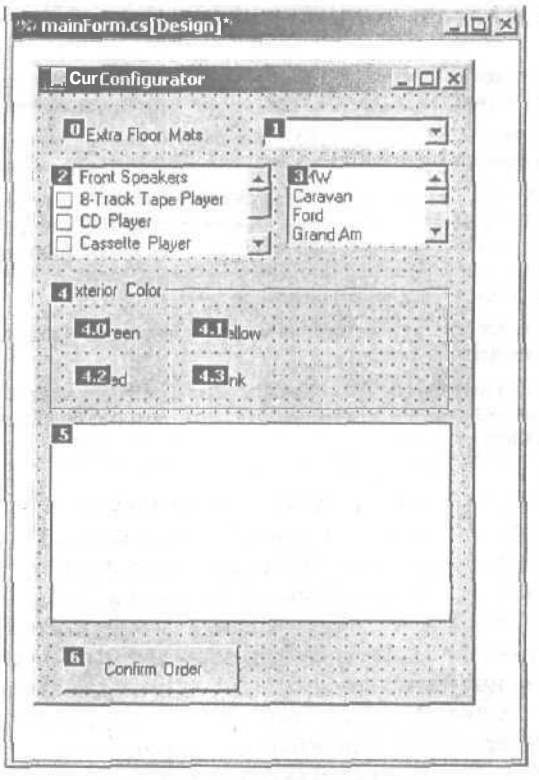


Рис. 10.16. Tab Order Wizard

Будем считать, что к этому моменту мы уже научились работать с наиболее распространенными элементами управления Windows. Однако .NET предлагает немало других, более экзотических элементов управления, с которыми мы познакомимся в следующих разделах.

Код приложения CarConfig можно найти в подкаталоге Chapter 10.

Элемент управления TrackBar

Элемент управления TrackBar (шкала с ползунком) позволяет пользователю производить выбор из диапазона возможных значений при помощи графических средств — ползунка, перемещаемого по шкале. Очень многие возможности TrackBar совпадают с возможностями обычной полосы прокрутки — ScrollBar. При работе с TrackBar нам потребуется настроить максимальное и минимальное значения, максимальный и минимальный шаг изменения, а также исходное положение ползунка. Все эти параметры устанавливаются при помощи свойств, представленных в табл. 10.13.

Таблица 10.13. Свойства класса TrackBar

Свойство	Назначение
LargeChange	Насколько передвинется ползунок при «значительных» изменениях — например, когда пользователь щелкнет мышью на шкале или воспользуется клавишами Page Up или Page Down
Maximum Minimum	Верхняя и нижняя границы шкалы TrackBar
Orientation	Расположение TrackBar Используются значения из перечисления Orientation (шкалу можно расположить горизонтально или вертикально)
SmallChange	Насколько переместится ползунок при «небольших» изменениях — например, когда пользователь воспользуется клавишами со стрелками
TickFrequency	Определяет, сколько делений на шкале будет отображаться. К примеру, если диапазон возможных значений у вас — от 0 до 200, отображать на шкале длиной два дюйма все 200 делений будет не очень удобно. Если мы установим значение для TickFrequency равным 5, то будет выведено 40 делений (каждое соответствует 5 единицам шкалы)
TickStyle	Определяет стиль отображения шкалы и ползунка. Используются значения из перечисления TickStyle
Value	Наиболее важное свойство TrackBar. Позволяет получить или установить численное значение положения ползунка, которое будет использоваться в приложении

Проиллюстрируем применение TrackBar на специальном приложении. В нем будет три объекта TrackBar, для каждого из которых используется шкала со значениями от 0 до 255. При перемещении любого из ползунков будет происходить событие Scroll, в обработчике которого мы поместим код, использующий значения, полученные от объектов TrackBar, для создания нового объекта Color. При этом RGB-значение этого объекта Color (цвет) будет зависеть от положения ползунков. Таким образом, у нас получится простое приложение для выбора пользователем цвета (конечно же, в System.Windows.Forms предусмотрено специальное диалоговое окно для этих целей, но нас оно сейчас интересовать не будет). Графический интерфейс нашего приложения представлен на рис. 10.17.

Будем считать, что три объекта TrackBar (redTrackBar — для красного цвета, greenTrackBar — для зеленого и blueTrackBar — для синего) уже созданы. Мы начнем с настройки их параметров:

```
// Настройка параметров для blueTrackBar
blueTrackBar.TickFrequency = 5;
blueTrackBar.Location - new System.Drawing.Point (104. 200);
```

```
blueTrackBar.TickStyle - System.Windows.Forms.TickStyle.TopLeft;
blueTrackBar.Maximum = 255:
blueTrackBar.Scroll += new System.EventHandler (this.blueTrackBar_Scroll);
```

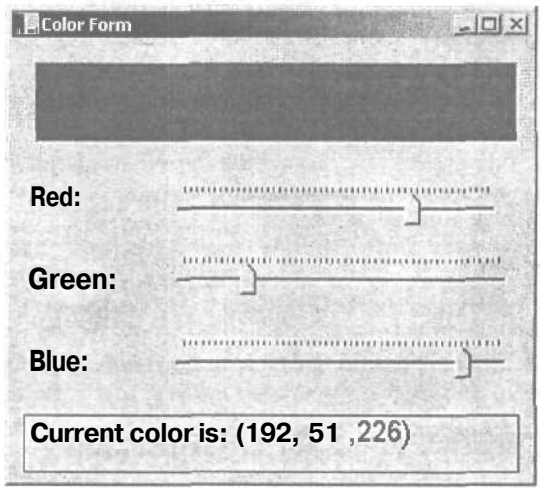


РИС. 10.17. Применение объектов TrackBar

Настройка всех остальных объектов TrackBar («красного» и «зеленого») производится точно так же, поэтому приводить соответствующий код мы не будем. Обратите внимание, что по умолчанию нижнее значение шкалы равно 0, и если это нас устраивает, то специально настраивать это значение совсем не обязательно. Код обработчика события Scroll для каждого из объектов TrackBar будет одинаковым, поэтому удобнее будет не дублировать этот код, а поместить в каждый из обработчиков событий ссылку на вспомогательную функцию:

```
protected void blueTrackBar_Scroll (object sender. System.EventArgs e)
{
        UpdateColor();
};
```

Функция UpdateColorO будет ответственна за выполнение двух главных действий: во-первых, за присвоение значений, полученных от объектов TrackBar, объекту Color (при помощи метода FromArgb()), а во-вторых, за изменение цвета фона объекта PictureBox, в котором в нашем случае никаких изображений не будет. Кроме того, мы еще позаботимся о выводе текстовой информации о выбранных значениях RGB в нижней части формы:

```
private void UpdateColorO
{
        // Получаем цвет
        Color c = Color.FromArgb(redTrackBar.Value, greenTrackBar.Value,
                                                    blueTrackBar.Value);

        // Изменяем фон объекта PictureBox (в нашем случае он называется colorBox)
        colorBox.BackColor = c;

        // Выводим текстовую информацию о выбранном цвете
```

```
        lblCurrColor.Text - "Current color is: " + "C" + redTrackBar.Value + ", " +
                            greenTrackBar.Value + " ," + blueTrackBar.Value + ")";
}
```

Последнее, что нам осталось сделать, — установить исходные значения для объектов TrackBar при загрузке формы. Выглядеть это может так:

```
public TrackForm()
{
        InitializeComponent()
        CenterToScreen();

        // Настраиваем исходное положение каждого ползунка
        redTrackBar.Value = 100;
        greenTrackBar.Value - 255;
        blueTrackBar.Value - 0;
        UpdateColor();
}
```

Код приложения Tracker можно найти в подкаталоге Chapter 10.

Элемент управления MonthCalendar

В пространстве имен System.Windows. Forms предусмотрен исключительно полезный элемент управления, при помощи которого пользователь может выбрать дату или диапазон дат, используя дружественный и удобный интерфейс. Речь идет об элементе управления MonthCalendar. Проиллюстрируем возможности этого элемента управления на примере нашего приложения CarConfig. Пусть пользователь получит возможность выбирать дату поставки автомобиля при помощи MonthCalendar. Новый интерфейс нашего приложения представлен на рис. 10.18.

Наиболее важные свойства MonthCalendar представлены в табл. 10.14.

Таблица 10.14. Свойства MonthCalendar

Свойство	Назначение
BoldedDates	Массив объектов DateTime, выделенных подсветкой
CalendarDimensions	Определяет количество выводимых строк и столбцов
FirstDayOfWeek	Определяет, с какого дня будет начинаться неделя в MonthCalendar
MaxDate	Самая поздняя дата, которую разрешается выбрать пользователю (по умолчанию ограничений нет)
MaxSelectionCount	Максимальное количество дат, которое одновременно может выбрать пользователь
MinDate	Самая ранняя дата, которую разрешается выбрать пользователю (по умолчанию ограничений нет)
MonthlyBoldedDates	Массив выделенных подсветкой объектов DateTime для месяца
SelectionEnd	Самая поздняя дата в диапазоне выделенных объектов
SelectionRange	Диапазон выделенных объектов
SelectionStart	Самая ранняя дата в диапазоне выделенных объектов
ShowToday ShowTodayCircle	Определяет, будет ли MonthCalendar выводить информацию о текущей дате в нижней части и выделять ее в календаре обводкой

Свойство	Назначение
ShowWeekNumbers	Определяет, будет ли MonthCalendar отображать номера недель справа от каждой строки
TodayDate	Дата, которая будет считаться MonthCalendar сегодняшней. По умолчанию TodayDate — это системная дата на момент создания объекта MonthCalendar
TodayDateSet	Определяет, можно ли пользователю по своему усмотрению выбирать сегодняшнюю дату. Если для этого свойства установлено значение true, пользователь может выбрать в качестве сегодняшней (TodayDate) любое число

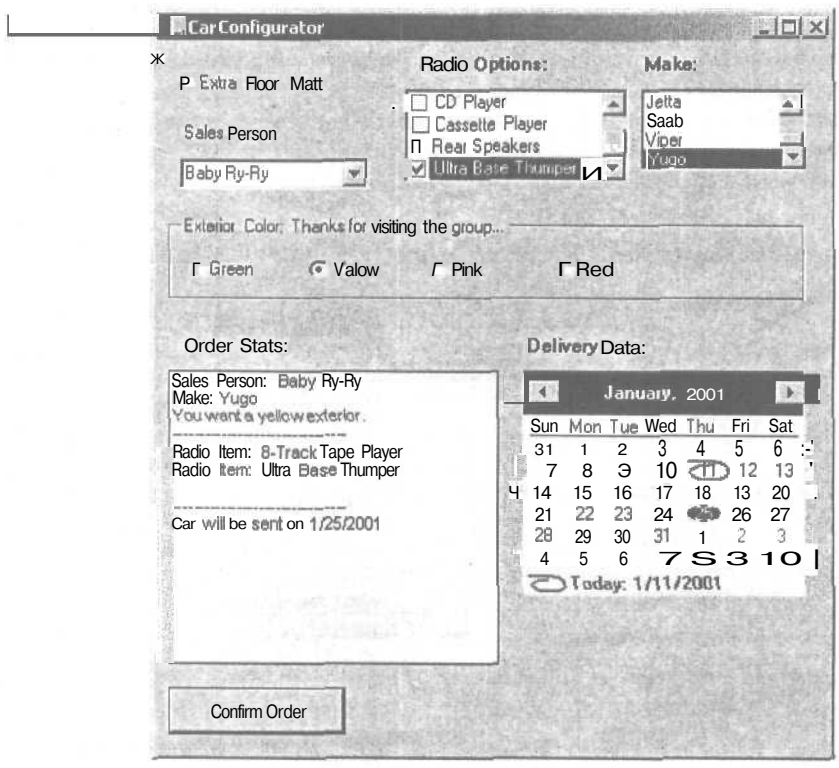


Рис. 10.18. Элемент управления MonthCalendar на форме

Несмотря на множество возможностей, работать с MonthCalendar очень проста. По умолчанию всегда выделяется (и подсветкой, и обводкой) текущая дата. Конечно же, пользователь может выбрать другую дату — в этом и есть смысл графического интерфейса MonthCalendar. Чтобы получить дату, выбранную пользователем, следует произвести обновление в обработчике события Click кнопки btnOrder:

```
protected void btnOrder_Click (object sender, System.EventArgs e)
{
        // Создаем переменную для хранения информации
        string orderInfo = "";
        ...
```

```
// Получаем выбранную пользователем дату поставки
DateTime d = monthCalendar.SelectionStart;
string dateStr = d.Month + " / " + d.Day + " / " + d.Year;
orderInfo += "Car will be sent: " + dateStr;
...
```

Можно получить дату, выбранную пользователем в MonthCalendar, при помощи свойства SelectionStart. Это свойство возвращает ссылку на объект DateTime, которую мы храним в специальной переменной (d). При помощи набора свойств типа DateTime мы можем извлечь всю необходимую информацию в нужном нам формате (к слову, мы можем получить не только дату, но и время, но это нам пока не нужно).

В нашем приложении мы разрешили пользователю выбирать только одну дату (то есть только один день) поставки автомобиля. Элемент управления MonthCalendar может быть использован и для выбора пользователем диапазона дат (например, автомобиль должен быть поставлен в период с такого-то числа по такое-то). Пользователю для выбора нескольких значений достаточно провести указателем при нажатой клавише мыши сразу по нескольким значениям (рис. 10.19).

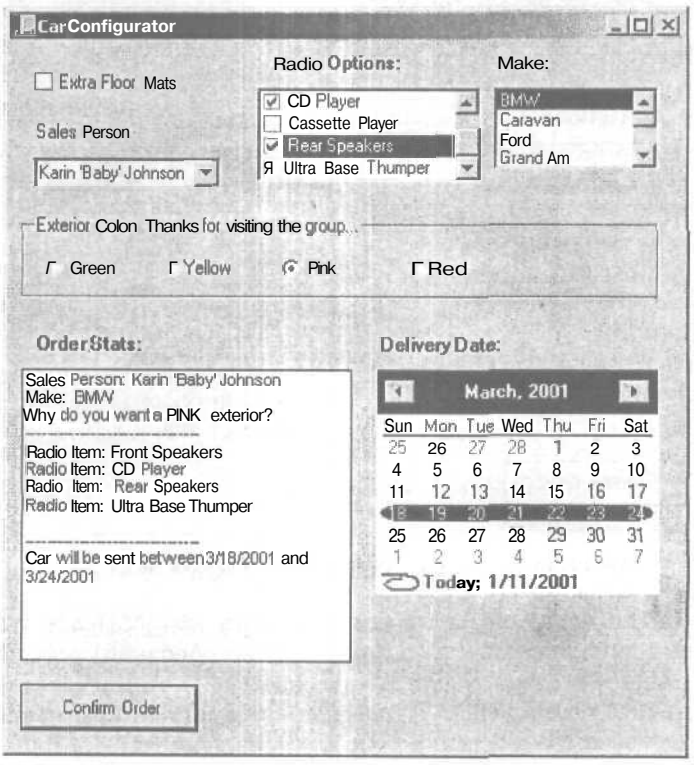


Рис. 10.19. Выбор нескольких дат в элементе управления MonthCalendar

Получение информации о выбранном диапазоне дат также не сложно. Для этого можно использовать свойства SelectionStart и SelectionEnd:

```
protected void btnOrder_Click (object sender. System.EventArgs e)
{
        // Создаем переменную для хранения информации
        string orderInfo = "";
        ...

        // Получаем диапазон дат. между которыми будет произведена поставка
        DateTime startD = monthCalendar.SelectionStart;
        DateTime endD = monthCalendar.SelectionEnd;

        string dateStartStr = startD.Month + " / " + startD.Day + " / " + startD.Year;
        string dateEndStr = endD.Month + " / " + endD.Day + " / " + endD.Year;

        // Для типа DateTime предусмотрено использование перегруженных операторов
        if(dateStartStr !- dateEndStr)
        {
                orderInfo +- "Car will be sent between " + dateStartStr + "\and\n" +
                                                                        dateEndStr;
        }
        else   // То есть выбрано одно число
                orderInfo += "Car will be sent on " + dateStartStr:
        ...
}
```

Еще немного о типе DateTime

В нашем примере мы извлекли выбранный пользователем диапазон дат в MonthCalendar при помощи свойств SelectionStart и SelectionEnd:

```
// Получаем два объекта DateTime
DateTime startD = monthCalendar.SelectIonStart;
DateTime endD = monthCalendar.SelectionEnd;
```

Затем при помощи свойств Month, Day и Year мы извлекли из объектов DateTime нужную нам информацию и сформировали текстовые строки. Это вполне допустимый подход (в том, что он работает, мы убедились на примере), но не самый удобный. Дело в том, что дату в необходимом текстовом формате проще получить из DateTime при помощи специальных «форматирующих» свойств самих объектов DateTime. Набор таких свойств (и некоторые методы) представлен в табл. 10.15.

Таблица 10.15. Члены класса DateTime

Член	Назначение
Date	Позволяет получить информацию о дате (дата всегда отсчитывается от полуночи)
Day Month Year	Позволяют получить соответственно день, месяц и число из текущего объекта DateTime
DayOfWeek	Возвращает день недели для объекта DateTime
DayOfYear	Возвращает номер дня в году для объекта DateTime
Hour Minute Second Millisecond	Возвращают информацию о часе, минутах, секундах и миллисекундах для объекта DateTime

продолжение

Таблица 10.15 (*продолжение*)

Член	Назначение
MaxValue MinValue	Возвращают минимальное и максимальное значения для DateTime
Now Today	Эти два *статических* свойства типа DateTime позволяют получить информацию о текущей дате и времени (Now) или только о текущей дате (Today)
Ticks	Позволяет получить счетчик «тиков» (с интервалом в 100 наносекунд) для объекта DateTime
ToLongDateString() ToLongTimeString() ToShortDateString() ToShortTimeString()	Преобразуют текущее значение объекта DateTime в разные виды текстового представления

При помощи вышеперечисленных членов мы сможем значительно упростить вывод текстовой информации о дате поставки автомобиля (собственно в результатах работы программы изменений не произойдет):

```
// Используем возможности класса DateTime
string dateStartStr - startD.Month + " / " + startD.Day + " / " startD.Year;
string dateStartStr * endD.Month + " / " + endD.Day + " / " endD.Year;
```

Элементы управления UpDown

В Windows.Forms предусмотрены еще два интересных элемента управления: DomainUpDown и NumericUpDown (иногда их также называют «вертушками» — spin control). По своему назначению эти элементы управления ближе всего к обычным (ListBox) и комбинированным (ComboBox) спискам: они также позволяют пользователю выбрать одно из возможных значений. Разница заключается в графических элементах: при использовании DomainUpDown и NumericUpDown пользователи выбирают нужное значение при помощи маленьких стрелок вверх и вниз. Пример применения этих элементов управления на форме представлен на рис. 10.20.

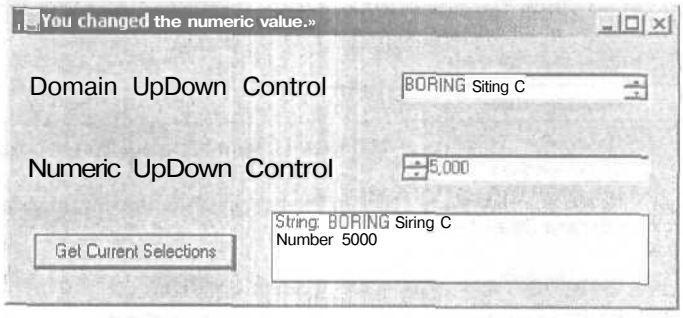


Рис. 10.20. Элементы управления DomainUpDown и NumericUpDown

Работа с «вертушками» ничуть не сложнее работы с ранее рассмотренными элементами управления. Как обычно, вся функциональность обеспечивается набором свойств. У NumericUpDown (используется для числовых значений) и DomainUpDown (для

значений текстовых) существует общий базовый класс — UpDownBase, от которого они наследуют большинство своих свойств. Члены UpDownBase представлены в табл. 10.16.

Таблица 10.16. Свойства класса UpDownBase

Свойство	Назначение
InterceptArrowKeys	Позволяет определить, разрешено ли пользователю для выбора значений использовать клавиши «вверх» и «вниз».
Readonly	Определяет, сможет ли пользователь найти нужное значение путем ввода в поле элемента управления или ему придется ограничиться выбором значений при помощи клавиш «вверх» и «вниз» или мыши
Text	Позволяет получить или установить текущий текст, отображаемый в «вертушке»
TextAlign	Позволяет определить выравнивание для текста в «вертушке»
UpDownAlign	Определяет, с какой стороны «вертушки» будут расположены кнопки «вверх» и «вниз». Используются значения из перечисления LeftRightAlignment

Для DomainUpDown предусмотрено несколько собственных свойств, которые обеспечивают доступ к текстовым значениям данного элемента управления. Эти свойства представлены в табл. 10.17.

Таблица 10.17. Свойства элемента управления DomainUpDown

Свойство	Назначение
Items	Позволяет получить доступ к набору текстовых значений внутри элемента управления DomainUpDown
SelectedIndex	Возвращает индекс (номер) выбранного в настоящий момент элемента
SelectedItem	Возвращает выбранное значение (а не его индекс)
Sorted	Определяет, будут ли текстовые значения упорядочены по алфавиту
Wrap	Определяет, будут ли текстовые значения «закольцованы», когда после последнего значения вновь следует первое

Свойства элемента NumericUpDown (который предоставляет пользователю выбор из набора числовых значений) представлены в табл. 10.18.

Таблица 10.18. Свойства элемента управления NumericUpDown

Свойство	Назначение
DecimalPlaces ThousandsSeparator Hexadecimal	Определяют формат отображения числового значения
Increment	Позволяет определить приращение для элемента управления. На величину этого приращения будет увеличиваться или уменьшаться значение в NumericUpDown при нажатии пользователем на соответствующие кнопки. По умолчанию равно 1
Minimum Maximum	Определяет верхнюю и нижнюю границы допустимых значений в NumericUpDown
Value	Позволяет получить или установить текущее значение NumericUpDown

Пример работы с DomainUpDown и NumericUpDown (для приложения, интерфейс которого был представлен на рис. 10.20) приведен ниже:

```
// Настраиваем параметры для элемента управления DomainUpDown
domainUpDown.Sorted = true;
domainUpDown.Wrap = true;
domainUpDown.Items.AddRange( new object[4] ("Another boring Sring named B", "Boring
String A", "BORING String C", "Final Boring String (D)"});
domainUpDown.SelectedIndex = 2;

// Настраиваем элемент управления NumericUpDown
numericUpDown.Maximum = new decimal (5000);
numericUpDown.ThousandsSeparator = true;
numericUpDown.UpDownAlign = LeftRightAlignment.Left;
```

Обработчик события Click для единственной кнопки нашего приложения просто извлекает информацию о выбранных пользователем в DomainUpDown и NumericUpDown значениях и выводит их на форме при помощи элемента управления Label:

```
protected void btnGetSelections_Click (object sender. System.EventArgs e)
{
        // Выводим выбранные пользователем значения
        lblCurrSel.Text = "String: " + domainUpDown.Text + "\n" +"Number: " +
                                                        numericUpDown.Value;
}
```

Конечно же, и DomainUpDown, и NumericUpDown поддерживают наборы событий. К примеру, если мы хотим определить какое-либо действие в ответ на выбор пользователем нового значения в «вертушке», в нашем распоряжении события SelectedItemChanged (для DomainUpDown) и ValueChanged (для NumericUpDown). **Пример приведен ниже:**

```
// Перехватываем событие SelectedItemChanged
DomainUpDown.SelectedItemChanged += newEventHandler (domainUpDown_SelectedItemChanged);
...

// Реагируем на событие
protected void domainUpDown_SelectedItemChanged (object sender, System.EventArgs e)
{
        this.Text = "You changed the string value...";
}
```

Код приложения UpAndDown можно найти в подкаталоге Chapter 10.

Элемент управления Panel

Мы уже работали с элементом управления GroupBox (группирующая рамка), при помощи которого можно логически объединить набор других элементов управления (например, переключателей). Назначение элемента управления Panel (панель) очень похоже: с его помощью мы также можем объединить прочие элементы управления на форме. Главное различие между Panel и GroupBox заключается в том, что Panel происходит от базового класса ScrollableControl и поэтому поддерживает полосы прокрутки. Менее существенное отличие заключается в том, что встроенного заголовка для Panel (в противоположность GroupBox) не предусмотрено.

Элементы управления Panel обычно используются для экономии пространства на форме. Например, если элементы управления, которые мы планируем разместить на форме, на ней не умещаются, мы можем поместить их внутрь Panel и установить для свойства AutoScrol 1 объекта Panel значение true. В результате пользователь получит возможность доступа к «не вмещающимся» элементам управления с помощью полос прокрутки. Давайте рассмотрим применение элемента Panel на примере нашего приложения TrackBar. Предположим, что мы решили сэкономить место и поместили все элементы управления TrackBar внутрь панели (рис. 10.21).

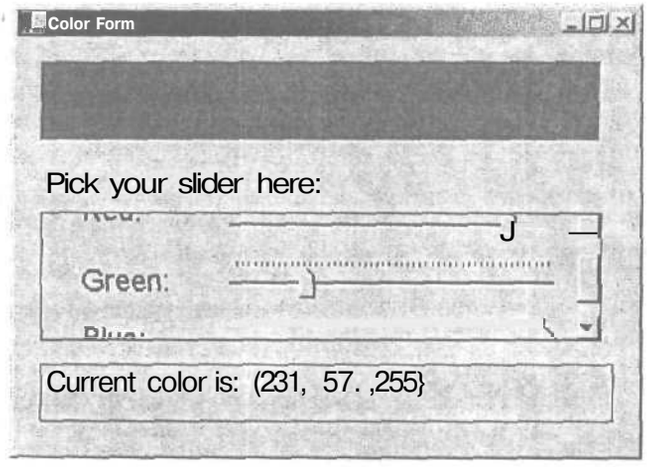


Рис. 10.21. Внутри элемента управления Panel можно размещать другие элементы управления

Код для работы с Panel выглядит практически одинаково с кодом для GroupBox. Наша задача — создать объект Panel (он у нас будет называться panel 1), настроить его свойства и добавить в него другие элементы управления при помощи свойства Controls:

```
// Настраиваем панель
panel1.AutoScroll * true;
panel1.Controls.Add (this.label2);
panel 1.Controls.Add (this.blueTrackBar);
panel1.Controls.Add (this.label3);
panel 1.Controls.Add (this.greenTrackBar);
panel1.Controls.Add (this.redTrackBar);
panel1.Controls.Add (this.label1);
```

Всплывающие подсказки (ToolTips) для элементов управления

Большинство приложений с современным пользовательским интерфейсом поддерживают всплывающие подсказки. В приложениях .NET эта возможность реализуется при помощи типа System.Windows.Forms.Tool T1 p. Можно сказать, что Tool Ti p (всплывающие подсказки) — это небольшие окна с текстом, появляющиеся при наведении указателя мыши на элемент управления на форме. Наиболее важные члены класса Tool Ti p представлены в табл. 10.19.

Таблица 10.19. Члены класса ToolTip

Член	Назначение
Active	Определяет, будет ли всплывающая подсказка активной. Возможность отключить всплывающие подсказки может быть полезной, например, если в приложении предусмотрено два варианта интерфейса: для обычных и для опытных пользователей
AutomaticDelay	Позволяет получить или установить время задержки (в миллисекундах) при появлении подсказки
AutoPopDelay	Время (в миллисекундах), в течение которого подсказка остается видимой, если указатель мыши неподвижен и находится в области, занимаемой соответствующим элементом управления. По умолчанию это значение равно 10 значениям AutomaticDelay
GetTooltip()	Возвращает текст подсказки
InitialDelay	Время (в миллисекундах), в течение которого указатель должен оставаться неподвижным в соответствующей области для появления подсказки. Значение по умолчанию равно значению AutomaticDelay
ReshowDelay	Время (в миллисекундах), в течение которого появится другая подсказка при перемещении указателя мыши от одного элемента управления к другому. По умолчанию это значение равно 1/5 от значения AutomaicDelay
SetToolTip()	Ассоциирует подсказку с элементом управления

Давайте добавим всплывающую подсказку к элементу управления MonthCalendar в нашем приложении CarConfig.To, что должно получиться, представлено на рис. 10.22,

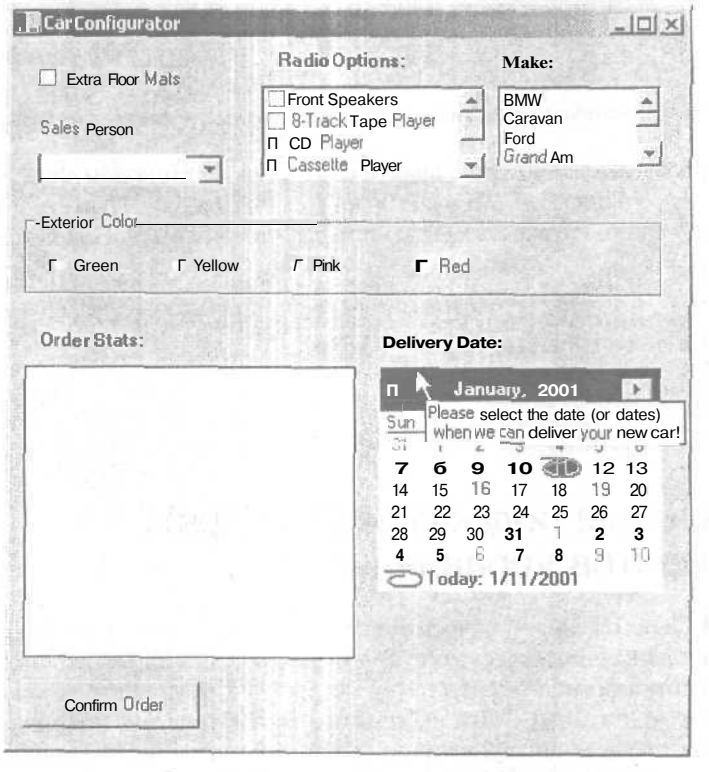


Рис. 10.22. Всплывающая подсказка

План действий — такой же, как и для любого другого элемента управления. Вначале мы создаем объект класса ToolTip (у нас он будет называться calendarTip), затем мы настраиваем его свойства и привязываем его к элементу управления на форме. Обратите внимание, что настройка текста подсказки и связывание подсказки с элементом управления производится в рамках единственного вызова метода SetToolTip():

```
// Настраиваем подсказку и связываем ее с элементом управления MonthCalendar
calendarTip.Active = true;
calendarTip.SetToolTip (monthCalendar, "Please select the date (or dates)\n when we car
                                                    deliver your new car!");
```

Добавление всплывающей подсказки при помощи графических средств Visual Studio

Для того чтобы настроить использование всплывающих подсказок для элементов управления, совершенно необязательно вообще писать какой-либо код. Мы можем сделать это очень просто с помощью графических средств Visual Studio.

Первое, что необходимо сделать — добавить на форму объект ToolTip, выбрав его в ToolBox (рис. 10.23). Затем мы можем указать текст всплывающей подсказки для любого элемента управления на форме (в том числе и для самой формы) и,з окна свойств данного элемента (рис. 10.24).

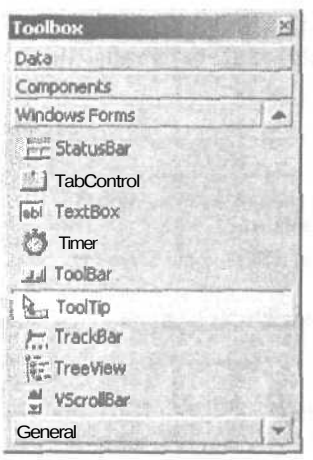


Рис. 10.23. Выбираем ToolTip в ToolBox и помещаем его на форму

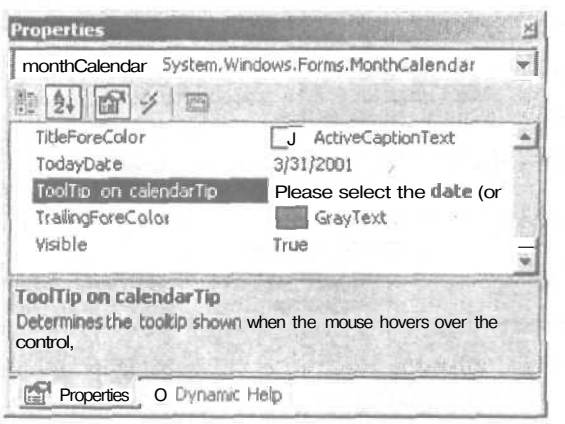


Рис. 10.24. Указываем текст всплывающей подсказки в окне свойств элемента управления

Элемент управления ErrorProvider

Очень часто возникает необходимость проверять в приложении данные, вводимые пользователем, на допустимость. Например, если пользователь выберет в диалоговом окне что-то не то, он должен быть об этом проинформирован (про создание диалоговых окон — дальше в этой главе).

Элемент управления ErrorProvider обеспечивает визуальное оповещение, сообщающее пользователю о недопустимости вводимых им данных. Например, пред-

ставим себе, что в нашем приложении предусмотрена форма с текстовым окном (объектом TextBox) и кнопкой (объектом Button). Пользователь не должен вводить в текстовое поле более 5 символов: если он попытается это сделать, он получит предупреждение (рис. 10.25).

Рис. 10.25. Элемент управления ErrorProvider

Когда обнаружена ошибка ввода данных пользователем, рядом с полем, в котором допущена ошибка, появляется маленький значок с восклицательным знаком. При наведении на него указателя мыши будет выведена подсказка с сообщением об ошибке. Кроме того, объект ErrorProvider можно настроить таким образом, чтобы значок несколько раз «мигнул», усиливая таким образом эффективность визуального оповещения.

Возможность проверки вводимых пользователем данных реализуется с помощью нескольких членов класса Control. Эти члены представлены в табл. 10.20.

Таблица 10.20. Некоторые члены класса Control

Член	Назначение
CausesValidation	Определяет, будет ли выбор этого элемента управления инициировать проверку данных в других элементах управления
Validated	Возникает при завершении проверки вводимых при помощи элемента управления данных
Validating	Возникает при начале проверки введенных пользователем данных (при выводе из фокуса данного элемента управления)

Для любого элемента управления на форме можно установить значение свойства CausesValidation равным true (по умолчанию для всех элементов управления оно имеет значение false). Если для элемента управления CausesValidation=true, то, когда он оказывается в фокусе, инициируется проверка введенных данных в остальных элементах управления (для которых такая проверка предусмотрена), то есть для каждого элемента управления инициируются события Validating и Validated. Именно в обработчике события Validating для элемента управления и стоит помещать код для работы с ErrorProvider. Кроме того, для выполнения различных действий по завершении проверки можно использовать обработчик события Validated.

Рассмотрим работу с ErrorProvider на примере нашего приложения с текстовым окном и кнопкой. Как обычно, вначале следует создать объект класса ErrorProvider, а затем настроить его свойства:

```
// Настраиваем свойства объекта ErrorProvider
errorProvider1.DataMember = "";
```

```
errorProvider1.DataSource = null;
errorProvider1.ContainerControl = null;
errorProvider1.BlinkStyle = System.Windows.Forms.ErrorBlinkStyle.AlwaysBlink;
errorProvider1.BlinkRate = 500;
```

В самом классе ErrorProvider предусмотрено совсем немного свойств. Для наших целей наибольший интерес представляет свойство BlinkStyle, для которого используются значения из перечисления ErrorBlinkStyle. Значения этого перечисления представлены в табл. 10.21.

Таблица 10.21. Значения перечисления ErrorBlinkStyle

Значение	Описание
AlwaysBlink	Значок с оповещением об ошибке будет «мерцать» постоянно: как при выводе сообщения о первой ошибке, так и при появлении нового описания ошибки
BlinkIfDifferentError	Значок с оповещением об ошибке будет «мерцать» только в ситуации, когда значок был уже выведен ранее, но текстовое описание ошибки изменилось
NeverBlink	Значок с оповещением об ошибке «мерцать» не будет

В классе ErrorProvider предусмотрены и другие важные члены. Например, если мы хотим использовать собственный значок, то это можно сделать при помощи свойства Icon. Привязка объекта ErrorProvider к элементу управления и настройка текста выводимой подсказки производится при помощи метода SetError() (обычно в обработчике события Validating для этого элемента управления):

```
protected void txtInput_Validating (object sender,
                                    System.ComponentModel.CancelEventArgs e)
{
        // Проверяем соответствие длины текста предельно допустимому значению
        //  (равному 5)
        if(txtInput.Text.ToString().Length > 5)
        {
                errorProvider1.SetError(txtInput, "Can't be greater than 5!");
        }
        else // Все в порядке, ничего показывать не будем
        errorProvider1.SetError(txtInput, "");
}
```

Код приложения ErrorProvider можно найти в подкаталоге Chapter 10.

Закрепление элемента управления в определенном месте формы

При создании формы, на которой будут расположены элементы управления, нам необходимо решить, будет ли пользователь иметь возможность изменять размеры этой формы или нет. В большинстве случаев размеры главного окна приложения можно изменять, а диалогового окна — нет. Настроить возможность (или невозможность) изменения размеров формы можно при помощи свойства Form.BorderStyle, для которого используются значения из перечисления FormBorderStyle (табл. 10.22).

Таблица 10.22. Значения перечисления FormBorderStyle

Значение	Описание
Fixed3D	Изменять размеры формы нельзя, форма выглядит «объемной»
FixedDialog	Изменять размеры формы нельзя, для формы используется толстая рамка (вид, типичный для диалоговых окон)
FixedSingle	Размеры формы изменять нельзя, форма обрамлена простой линией
RxedToolWindow	Форма будет лишена некоторых управляющих элементов в строке заголовка, размеры изменять нельзя
None	Рамки вокруг формы вообще не будет
Sizable	Стандартный вид формы главного окна приложения. Размеры формы можно изменять
SizableToolWindow	То же, что и RxedToolWindow, но размеры формы можно изменять

Предположим, что мы разрешили пользователям изменять размеры формы. Сразу возникает несколько интересных вопросов: а как поведут себя при этом элементы управления на форме? Например, что будет, если пользователь сделает форму меньше, чем необходимо для отображения всех элементов управления — будут ли элементы управления автоматически изменять свои размеры (а возможно, и местонахождение) или просто они скроются за пределами видимой части формы? Поведение элемента управления, при котором он будет привязан к определенному месту на форме, остающемуся неизменным при изменении ее размеров, определяется с помощью свойства Anchor. Для этого свойства используются значения из перечисления AnchorStyles (табл. 10.23).

Таблица 10.23. Значения перечисления AnchorStyles

Значение	Описание
Bottom	Элемент управления будет прикреплен к нижней части своего контейнера
Left	Элемент управления будет прикреплен к левой части своего контейнера
None	Элемент управления вообще не будет прикреплен
Right	Элемент управления будет прикреплен к правой части своего контейнера
Top	Элемент управления будет прикреплен к верхней части своего контейнера

Если, к примеру, нам потребовалось прикрепить элемент управления к верхнему левому углу, мы вправе применить сразу несколько стилей (AnchorStyles.Top и AnchorStyles.Left).

Таким образом, если мы прикрепим элемент управления к правому краю формы:

```
MyButton.Anchor - AnchorStyles.Right;
```

тем самым мы гарантируем, что при любом изменении размеров формы кнопка все равно будет находиться на ее правому крае.

Стыковка элемента управления с краем формы

Еще одна возможность разместить элемент управления возле края формы заключается в применении стыковки (docking). Как и в случае с закреплением (an-

choring), элемент управления будет размещаться у выбранного нами края формы вне зависимости от изменения ее размеров. Отличие стыковки от закрепления заключается в том, что при использовании стыковки элемент управления будет занимать все пространство у края формы, изменяя свои размеры при изменении размеров последней. Управление стыковкой производится при помощи свойства Dock. Возможные значения этого свойства (определенные при помощи перечисления DockStyle) представлены в табл. 10.24.

Таблица 10.24. Значения перечисления DockStyle

Значение	Описание
Bottom	Элемент управления займет место у нижнего края формы
Fill	Элемент управления будет пристыкован ко всем краям формы сразу и, таким образом, займет всю клиентскую область формы
Left	Элемент управления займет место у левого края формы
None	Элемент управления не будет пристыкован ни к одному из краев
Right	Элемент управления займет место у правого края формы
Top	Элемент управления займет место у верхнего края формы

Если нам потребуется обеспечить стыковку элемента управления с левым краем формы, мы можем использовать следующий код:

```
// Этот элемент управления всегда будет занимать место у левого края формы, независимо
// от ее размеров:
myButton.Dock = DockStyle.Left;
```

То, как эта кнопка будет Выглядеть на форме, представлено на рис. 10.26.

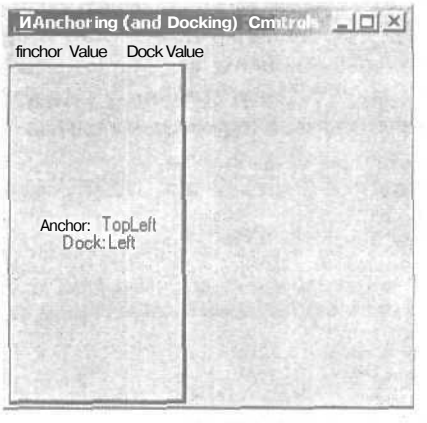


Рис. 10.26. Кнопка пристыкована к левому краю формы

Чтобы можно было наглядно продемонстрировать, как отражаются на элементе управления разные значения свойств Dock и Anchor, мы приготовили специальное приложение AnchoringControls (его можно найти в подкаталоге Chapter 10). Окно этого приложения представлено на рис. 10.27.

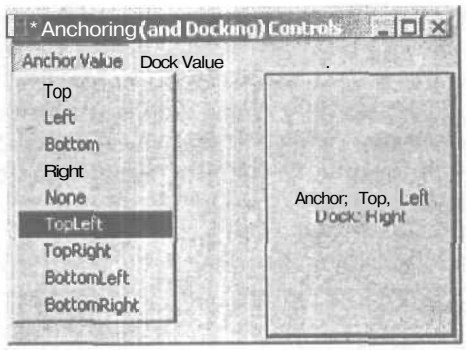


Рис. 10.27. Приложение для иллюстрации возможностей стыковки и закрепления

Создание пользовательских диалоговых окон

Теперь, когда мы вооружены знаниями о наиболее важных элементах управления, определенных в пространстве имен System.Windows.Forms, настало время обратиться к созданию пользовательских диалоговых окон. Практически все элементы управления, которые применяются в обычных формах, можно использовать и в диалоговых окнах.

Отдельный класс Dialog в .NET не предусмотрен. Диалоговое окно — это форма, обладающая некоторыми специальными характеристиками. Первая отличительная черта большинства диалоговых окон — то, что их размер изменять нельзя. Кроме того, в диалоговых окнах обычно не используются элементы управления, помещаемые в верхнюю часть обычных форм: ControlBox, MinimizeBox и MaximizeBox. Можно сказать, что для пользователя диалоговое окно в противоположность обычному является практически неизменяемым.

Для открытия модального диалогового окна (это такое диалоговое окно, которое нужно обязательно закрыть перед возвращением к исходной форме) используется метод ShowDialog(). Предположим, что диалоговое окно в нашем приложении открывается при щелчке пользователем на пункте в меню. Код для открытия диалогового окна может выглядеть следующим образом:

```
// Открываем модальное диалоговое окно
protected void mnuModalBox_Click (object sender. System.EventArgs e)
{
        SomeCustomForm myForm = new SomeCustomForm();

        // То же самое можно сделать и в конструкторе SomeCustomForm
        myForm.BorderStyle = FormBorderStyle.FixedDialog;
        myForm.ControlBox = false;
        myForm.MinimizeBox = false;
        myForm.MaximizeBox = false;

        // Передаем как ссылку родительской форме
        myForm.ShowDialog(this);

        DoSomeWork();
}
```

Обратите внимание, что за вызовом метода ShowDialog() у нас сразу следует вспомогательная функция DoSomeWork(). Модальность формы определяет именно метод

ShowDialog(): при использовании нашего кода весь ход выполнения программы будет приостановлен вплоть до того момента, пока ShowDialog() не вернет соответствующее значение. Для пользователя это значит, что ему придется закрыть диалоговое окно, прежде чем он сможет выполнить какие-либо операции на главной форме. Если для нашего диалогового окна модальность не нужна (то есть в фокус смогут попадать как диалоговое окно, так и главная форма), достаточно вместо ShowDialog() использовать метод Show(). Выглядеть это может так:

```
// Открываем немодальное диалоговое окно
protected void mnuShowMyDlg_Click (object sender, System.EventArgs e)
{
        SomeCustomForm myForm = new SomeCustomForm();
        myForm.BorderStyle = FormBorderStyle.FixedDialog;
        myForm.ControlBox = false;
        myForm.MinimizeBox = false;
        myForm.MaximizeBox = false;
        myForm.Show();

        DoSomeWork();
}
```

В этом случае немедленно после открытия диалогового окна начнется выполнение DoSomeWork() — безо всяких приостановок.

Пример использования диалогового окна в приложении

Спроектируем приложение, в котором будет использоваться пользовательское диалоговое окно. Приложение будет несложным: главная форма, имеющая меню с единственным пунктом (рис. 10.28). При активации этого пункта откроется модальное диалоговое окно (рис. 10.29).

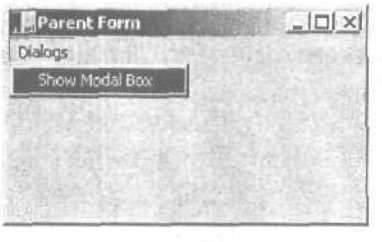


Рис. 10.28. Главная форма приложения

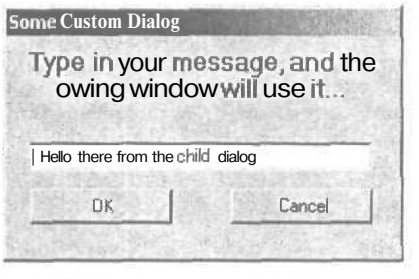


Рис. 10.29. Модальное диалоговое окно

В диалоговом окне пользователь может ввести текст, который при нажатии кнопки (Ж будет выведен на форме (рис. 10.30).

Рис. 10.30. Текст, введенный пользователем в диалоговом окне, выводится на форме

Кроме того, мы сделаем так, что если пользователь повторно откроет диалоговое окно, в его текстовое поле будет помещен текст, ранее введенный пользователем и отображенный на форме (рис. 10.31).

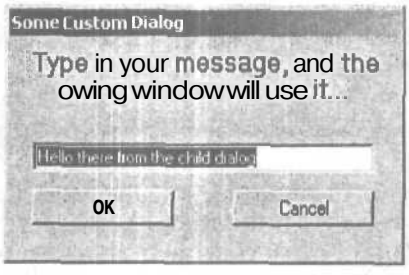


Рис. **10.31.** Диалоговое окно при повторном открытии

В коде для диалогового окна ничего принципиально нового для нас не встретится: как мы помним, диалоговое окно — это не более чем слегка измененный Form. Код может выглядеть так:

```
// Наше диалоговое окно
public class SomeCustomForm : System.Windows.Forms.Form
{
        private System.Windows.Forms.Button btnCancel;
        private System.Windows.Forms.Button btnOK;
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.TextBox txtMessage;

        public SomeCustomForm()
        {
                InitializeComponent();
                this.StartPosition = FormStartPosition.CenterParent;
        }
        ...

        private void InitializeComponent()
        {
                ...
                // Настраиваем кнопку OK
```

```
        btnOK.DialogResult = System.Widows.Forms.DailogResult.OK;
        btnOK.Size = new System.Drawing.Size (96, 24):
        btnOK.Text - "OK";

        // Настраиваем кнопку Cancel
        btnCancel.DialogResult = System.Windows.Forms.DialogResult.Cancel;
        btnCancel.Size - new System.Drawing.Size (96, 24);
        btnCancel.Text = "Cancel";

        // Настраиваем форму - диалоговое окно
        this.Text = "Some Custom Dialog";
        this.MaximizeBox = false;
        this.ControlBox - false;
        this.MinimizeBox = false:
        ...
    }
}
```

В конструкторе диалогового окна мы воспользовались свойством StartPosition, установив для него значение CenterParent из перечисления FormStartPosition. Ранее в аналогичных ситуациях мы просто вызывали для формы метод CenterToScreen(). Различие между этими двумя подходами заключается в том, что CenterToScreen() выводит форму по центру экрана, а CenterParent — по центру родительской формы. При работе с диалоговыми окнами более аккуратным представляется использование CenterParent.

Самый важный аспект программирования диалоговых окон заключается в назначении определенным кнопкам значений из перечисления DialogResult. Как вы, наверное, уже замечали, в большинстве диалоговых окон в самых разных приложениях предусмотрены кнопки OK и Cancel. Нажатием на кнопку OK пользователь как бы сообщает приложению: «Я все выбрал, и можно идти дальше и использовать в программе выбранные мною значения». Нажатие на кнопку Cancel означает: «Я раздумал работать с этим диалоговым окном, закройте его, пожалуйста». Назначение кнопкам соответствующих значений производится следующим образом:

```
private void InitializeComponent()
{
    ...
    // Назначения кнопке значения OK
    btnOK.DialogResult = System.Windows.Forms.DialogResult.OK;

    // Настройка кнопке значения Cancel
    btnCancel.DialogResult = System.Windows.Forms.DialogResult.Cancel;

    ...
}
```

Хорошо, мы назначили кнопкам значения OK и Cancel. Что это означает? Во-первых, при нажатии на эти кнопки *диалоговое окно будет автоматически закрыто,* Во-вторых, в ходе дальнейшего выполнения программы мы можем выяснить значение свойства DialogResult, чтобы узнать, какую кнопку нажал пользователь, и в зависимости от результата выполнить определенные действия:

```
protected void mnuModalBox_Click (object sender. System.EventArgs e)
{
```

```
// Создаем диалоговое окно
SomeCustomForm myForm - new SomeCustomFormO:

// Передаем ссылку
myForm.ShowDialog(this);

if(myForm.DialogResult - = DialogResult.OK)
i
        // Пользователь нажал OK! Делаем то. что должны сделать.
}

DoSomeWork():
```

I

Значения перечисления DialogResult представлены в табл. 10.25. Помните, что в коде самого диалогового окна эти значения можно присваивать кнопкам. В коде же для главной формы приложения мы сравниваем эти значения со значением свойства DialogResult для самого диалогового окна.

Таблица 10.25, Значения перечисления DialogResult

Значение	Описание
Abort	Значение, возвращаемое диалоговым окном — Abort (аварийное или внеплановое завершение). Это значение обычно присваивается специальной кнопке Abort
Cancel	Значение, возвращаемое диалоговым окном — Cancel (отмена). В большинстве диалоговых окон предусмотрена специальная кнопка Cancel
Ignore	Значение, возвращаемое диалоговым окном — Ignore (игнорировать, пропустить). Как правило, существует специальная кнопка Ignore в диалоговом окне
No	Значение, возвращаемое диалоговым окном — No (нет). И для этого значения обычно используется специальная кнопка
None	Из диалогового окна ничего не возвращается. Это означает, что модальное диалоговое окно все еще открыто
OK	Значение, возвращаемое диалоговым окном — OK. Для генерации этого значения обычно используется специальная кнопка OK
Retry	Значение, возвращаемое диалоговым окном — Retry (повторить). И для этого значения чаще всего предусматривается специальная кнопка
Yes	Значение, возвращаемое диалоговым окном — Yes (да). Обычно посылается при помощи кнопки Yes

Как получить данные из диалогового окна

Конечно же, данные из диалогового окна передаются при помощи переменных. В нашем примере мы выводили текст, введенный пользователем, на родительской форме диалогового окна. Мы слегка изменим наше приложение, добавив в него дополнительные переменные для хранения данных, вводимых пользователем:

```
public class SomeCustomForm : System.Windows.Forms.Form
{
        public SomeCustomFormO
        {
                InitializeComponent();
```

```
            this.StartPosition = FormStartPosition.CenterParent;
        }

        // Переменная для хранения данных, полученных из диалогового окна
        private string strMessage;

        public string Message
        {
            get {return strMessage; }

            // Код функции set позволит нам настраивать исходное значение
            // для текстового поля в диалоговом окне
            set
            {
                strMessage = value;
                txtMessage.Text - str.Message;
            }
        }
...
}
```

Для передачи значения из текстового поля в диалоговом окне нам придется также позаботиться о перехвате события Click кнопки OK. Помните, что назначение кнопке значения DialogResult.OK означает, что при нажатии на эту кнопку диалоговое окно будет закрыто. Однако в нашем случае в обработчике события Click нам потребуется выполнить дополнительные действия:

```
protected void btnOK_Click (object sender, System.EventArgs e)
{
    // Кнопка OK нажата! Настраиваем новые сообщения
    strMessage = txtMessage.Text;
}
```

Собственно говоря, мы сделали почти все. Конечно же, в нашем приложении диалоговое окно может использоваться для ввода пользователем значительно большего количества данных. В этом случае нам потребуются дополнительные свойства для сохранения информации о данных, введенных или выбранных пользователем в диалоговом окне, но принцип остается тем же самым. В нашем примере осталось только предусмотреть код для извлечения значения из диалогового окна и применения этого значения в приложении. Этот код может выглядеть следующим образом:

```
protected void mnuModalBox_Click (object sender, System.EventArgs e)
{
    // Создаем диалоговое окно
    SomeCustomForm myForm = new SomeCustomForm();

    // Будем считать что у нас уже есть переменная типа string, которая называется
    // dlgMsg
    myForm.ShowDialog(this);
    myForm.Message = dlgMsg;

    if(myForm.DialogResult = = DialogResult.OK)
    {
        dlgMsg = myForm.Message;
        Invalidate();
    }
```

```
        DoSomeWork();
}
```

Теперь мы можем делать с полученным значением все, что угодно. Например, вывести его стандартным способом в клиентской части формы:

```
protected void mainForm_Paint (object sender. PaintEventArgs e)
{
        // Выводим текстовую строку, полученную из диалогового окна
        Graphics g - e.Graphics;
        g.DrawString(dlgMsg, new Font("times New Roman", 24). Brushes.Blue,
                                                    this.ClientRectangle);
}
```

Код приложения SimpIeDialog можно найти в подкаталоге Chapter 10.

Наследование форм

Последнее, о чем необходимо рассказать в этой главе, — о наследовании форм в C#. Наследование — это один из столпов ООП, который позволяет одному классу расширять возможности другого класса. Обычно когда речь идет о наследовании, мы подразумеваем наследование одного типа без какого-либо графического интерфейса от другого такого же типа. Однако в мире Windows Forms мы вполне можем произвести один класс Form от другого класса Form, сохранив при этом и настроенные в базовом классе элементы управления, и все остальные возможности исходной формы.

Проиллюстрируем это на примере. Предположим, что мы поместили класс CarConf1 gForm.cs из наших предыдущих примеров в библиотеку кода C# (CarConfigLib) и откомпилировали соответствующий модуль DLL. После этого мы создали новый проект Windows Application. Будем считать, что наша задача — создать в этом проекте новый класс Form, производный от CarConfigFrom.cs.

Первое, что мы должны сделать — добавить ссылку на сборку CarConfigLib.dll. Следующий этап — определить отношения наследования между двумя классами стандартными средствами C#:

```
// Используем пространство имен базового класса
using CarConfig;

// Созданная нами форма - это класс, производный от CarConfigForm
public class DerivedForm : CarConfig.CarConfigForm
{...};
```

Если мы создадим экземпляр типа DerivedForm в приложении Windows Form, то убедимся, что DerivedForm унаследовал все элементы управления от CarConfigForm! Весь необходимый для отображения унаследованных элементов управления код находится в определении метода InitializeComponent(). Все элементы управления, которые были объявлены в базовом классе как Private, не могут быть перемещены в производном классе на другое место. Если же мы поменяем определение класса CarConfigForm таким образом, что элементы управления станут protected вместо private, то сможем изменять их местонахождение при помощи графических средств конструирования формы. Конечно же, мы можем расширять функциональность производного класса, например, путем добавления своих элементов управления.

Например, для наших целей мы можем добавить новое главное меню с единственным пунктом для выхода из приложения (рис. 10.32).

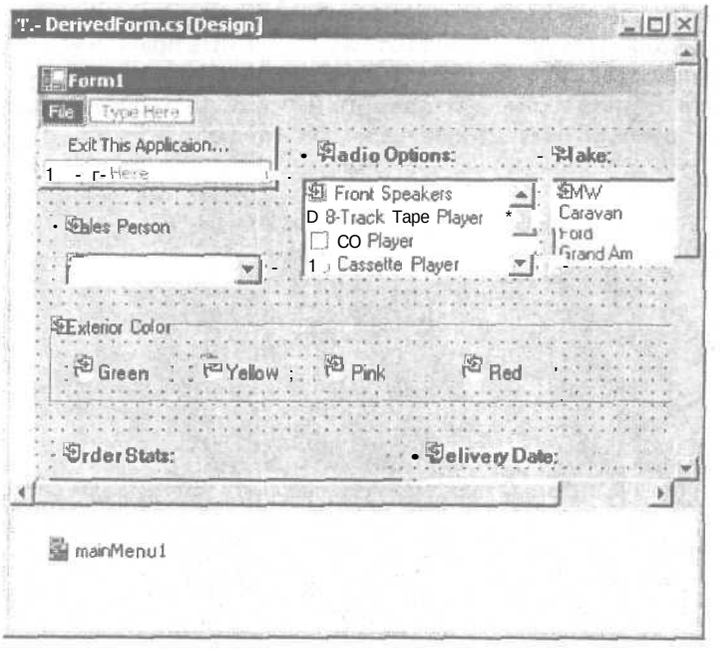


Рис. 10.32. Мы добавили в производную форму новое главное меню

Обработчик события Click для единственного пункта нашего меню будет очень простым:

```
private void mnuFileExit_Click(object sender, System.EventArgs e)
{
      this.Close();
}
```

Полезно будет упомянуть, что в Visual Studio.NET IDE предусмотрен специальный мастер для создания производных форм. Для его запуска достаточно выбрать в меню Project (Проект) пункт Add Inherited Form (Добавить унаследованную форму). Для создания производного класса нам вначале нужно будет указать имя этого создаваемого класса, а затем выбрать сборку с базовым классом и в нем — нужный базовый класс-форму.

Код приложений MyDerivedForm и CarConfigLib можно найти в подкаталоге Chapter 10.

Подведение итогов

Эта глава была посвящена работе с элементами управления Windows — от самых простых и распространенных (таких как Button) до тех, которые можно с полным правом назвать экзотическими (MonthCalendar). Конечно же, мы смогли подробно

рассмотреть далеко не все существующие элементы управления — кое-что вы должны будете освоить самостоятельно. Однако общие принципы работы с элементами управления остаются теми же.

Мы также рассмотрели возможности закрепления и стыковки элементов в разных местах формы, которые позволяют нам обеспечить правильное расположение элементов управления при изменении размеров формы.

В заключительной части этой главы мы научились создавать диалоговые окна и работать с ними, а также производить новые формы от уже существующих форм.

Ввод, вывод и сериализация объектов

11

При создании полноценных приложений обычно необходимо обеспечить возможность сохранять результаты трудов пользователей в перерывах между сеансами работы. В этой главе будут рассмотрены темы, связанные с реализацией ввода-вывода данных в .NET. Вначале мы рассмотрим наиболее важные типы, определенные в пространстве имен System.IO, и то, как можно программным образом работать со структурой файлов и каталогов на диске. После этого мы познакомимся с разнообразными способами чтения и записи символьных, двоичных и строковых данных из тех самых мест, где они могут храниться, а также непосредственно из оперативной памяти.

Вторая часть этой главы посвящена особенностям процессов сериализации в .NET. Сериализация — это процесс преобразования состояния объекта (или набора взаимосвязанных объектов) в специальное представление (например, в формате XML), которое может быть помещено в поток (например, для записи в файл) и затем восстановлено из него. Мы поработаем с атрибутами [Serializable] и [NonSerialized], а также научимся многим возможностям, связанным с сериализацией путем реализации интерфейса ISerializable.

В самом конце главы мы проиллюстрируем итоги обсуждения при помощи приложения Windows Forms, в рамках которого пользователь сможет работать с набором объектов Car, сохраняя информацию об этих объектах на диске и восстанавливая их с диска. Мы также научимся работать с объектом DataGrid, который будет постоянно использоваться в главе 13, посвященной ADO.NET.

Знакомство с пространством имен System.IO

Пространство имен System.IO содержит в себе большой набор типов, которые предназначены для выполнения операций с файлами и другими операциями ввода и вывода. Все типы System.IO — классы, перечисления, структуры и делегаты находятся в библиотеке mscorlib.dll. Часть типов System.IO представлена на рис. 11.1 в окне ILDasm.exe.

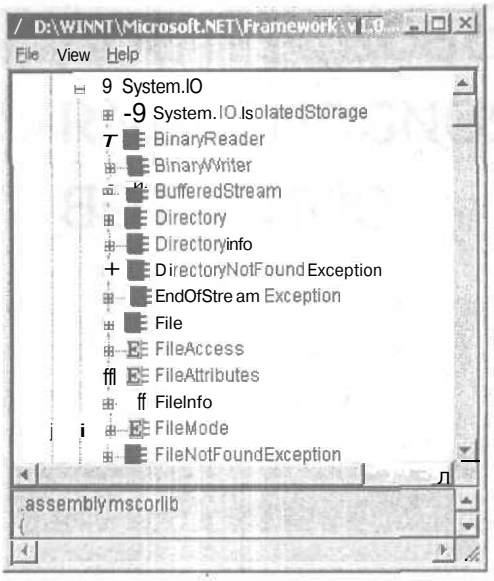


Рис. 11.1. Пространство имен System.IO

Как мы увидим, большинство классов System.IO предназначено для работы с каталогами и файлами на диске. Однако есть и такие типы, которые позволяют работать с буферами в оперативной памяти или с областями оперативной памяти напрямую. Наиболее важные неабстрактные классы System.IO представлены в табл. 11.1.

Таблица 11.1. Наиболее важные классы пространства имен System.IO

Класс	Описание
BinaryReader BinaryWriter	Позволяют сохранять и извлекать информацию типов данных-примитивов (целочисленных, логических, строковых и т. п.) как двоичные значения
BufferedStream	Обеспечивает временное хранилище для потока байтов (например, для последующего переноса в постоянное хранилище)
Directory DirectoryInfo File FileInfo	Используются для работы со свойствами указанного каталога или физического файла, а также для создания новых файлов и расширения существующей структуры каталогов. Возможности классов File и Directory реализованы главным образом в виде статических методов. Классы DirectoryInfo и FileInfo работают через обычные объекты данных классов
FileStream	Обеспечивает произвольный доступ к файлу, представляемому как поток байтов
MemoryStream	Также обеспечивает произвольный доступ к потоку байтов, но уже не в виде физического файла, а в оперативной памяти
StreamWriter StreamReader	Используются для считывания из файла или записи в файл текстовой информации. Произвольный доступ к файлам при помощи этого класса не поддерживается
StringWriter StringReader	Эти классы также предназначены для работы с текстовой информацией, однако они применяются для работы с буфером в оперативной памяти, а не с файлом на диске

Помимо обычных классов, представленных в табл. 11.1, в System.IO предусмотрено также большое количество перечислений и абстрактных классов. Абстрактные классы (Stream, TextReader, TextWriter и т. п.) определяют общие полиморфические интерфейсы для производных классов. С многими из этих абстрактных и производных классов мы познакомимся подробнее в этой главе.

Типы Directory(Info) и File(Info)

В пространстве имен System.IO предусмотрено четыре класса, которые предназначены для работы с физическими файлами на диске и структурой каталогов на диске. Первые два типа — Directory и File — позволяют выполнять операции в файловой системе (создание, удаление и т. п.) при помощи статических членов. DirectoryInfo и FileInfo обладают схожими функциональными возможностями, но они реализуются путем создания объектов данных классов. Иерархия этих классов представлена на рис. 11.2. Обратите внимание, что типы Directory и File напрямую производятся от System.Object, а DirectoryInfo и FileInfo происходят от абстрактного класса FileSystemInfo.

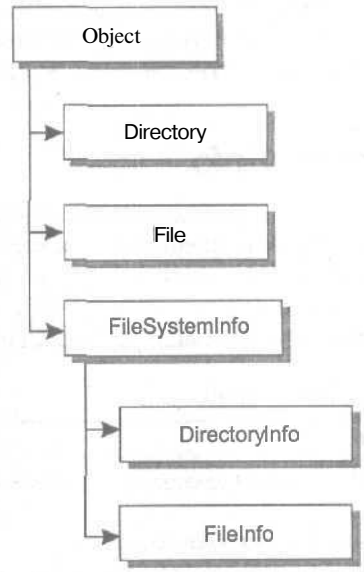


Рис. 11.2. Иерархия классов для работы с файлами и каталогами

Абстрактный класс FileSystemInfo

Классы DirectoryInfo и FileInfo наследуют значительную часть своих возможностей от абстрактного класса FileSystemInfo. Значительная часть членов FileSystemInfo предназначена для работы с общими характеристиками файла или каталога (метками времени, атрибутами и т. п.). Наиболее интересные свойства FileSystemInfo представлены в табл. 11.2.

Таблица 11.2. Свойства класса FileSystemInfo

Свойство	Описание
Attributes	Позволяет получить или установить атрибуты для данного объекта файловой системы. Для этого свойства используются значения и перечисления FileAttributes
CreationTime	Позволяет получить или установить время создания объекта файловой системы
Exists	Может быть использовано для того, чтобы определить, существует ли данный объект файловой системы
Extension	Позволяет получить расширение для файла
FullName	Возвращает имя файла или каталога с указанием пути к нему в файловой системе
LastAccessTime	Позволяет получить или установить время последнего обращения к объекту файловой системы
LastWriteTime	Позволяет получить или установить время последнего внесения изменений в объект файловой системы
Name	Возвращает имя указанного файла. Это свойство доступно только для чтения. Для каталогов возвращает имя последнего каталога в иерархии, если это возможно. Если нет, возвращает полностью определенное имя

В FileSystemInfo предусмотрены и несколько методов. Например, метод Delete() позволяет удалить объект файловой системы с жесткого диска, а Refresh() — обновить информацию об объекте файловой системы, например, перед обращением к его атрибутам.

Работа с типом DirectoryInfo

Первый «нормальный» класс, с которым мы познакомимся, — класс DirectoryInfo. Он содержит набор членов, которые предназначены для создания, перемещения, удаления, получения информации о каталогах и подкаталогах в файловой системе. Помимо членов, унаследованных от FileSystemInfo, DirectoryInfo определяет также набор своих собственных, уникальных членов, представленных в табл. 11.3.

Таблица 11.3. Члены класса DirectoryInfo

Член	Описание
Create() CreateSubDirectory()	Создают каталог (или подкаталог) по указанному пути в файловой системе
Delete()	Удаляет каталог со всем его содержимым
GetDirectories()	Возвращает массив строковых значений, представляющих все подкаталоги
GetFiles()	Позволяет получить файлы в текущем каталоге (в виде массива объектов FileInfo)
MoveTo()	Перемещает каталог и все его содержимое на новый адрес в файловой системе
Parent	Возвращает родительский каталог в иерархии файловой системы

Работа с типом DirectoryInfo начинается с того, что мы указываем путь к данному каталогу (например, C:\, D:\WinNT, \\CompnayServer\Utils, A:\ и т. п.) как параметр для конструктора объекта DirectoryInfo. Если мы хотим обратиться к текущему

каталогу (то есть каталогу, в котором в настоящее время производится выполнение приложения), используем обозначение ".". Вот несколько примеров:

```
// Создаеи объект DirectoryInfo, которому будет соответствовать текущий каталог
DirectoryInfo dirl = new DirectoryInfo(".");

// Создаеи объект DirectoryInfo, которому будет соотвествовать каталог C:\Foo\Bar
DIrectoryInfo dir2 = new DirectoryInfo(@"C:\Foo\Bar");
```

Если мы попытаемся создать объект DirectoryInfo для несуществующего каталога, будет сгенерировано исключение System.IO.DirectoryNotFoundException. Если же все нормально, то мы сможем получить информацию о данном каталоге и его содержимом при помощи свойств DirectoryInfo. В примере, который приведен ниже, мы создаем объект DirectoryInfo, которому соответствует каталог D:\WinNT (измените этот путь, если на вашем компьютере файлы Windows 2000 лежат в другом месте), и выводим информацию о каталоге файловой системы:

```
class MyDirectory
{
        public static void Main(String[] args)
        {
                // Создаем объект DirectoryInfo, соответствующий D:\WinNT
                DirectoryInfo dir - new DirectoryInfo(@"D:\WinNT");

                // Выводим информацию о каталоге
                Console.WriteLine("***** Directory Info *****");
                Console.WriteLine("FullName: {0}", dir.FullName);
                Console.WriteLine("Name: {0}", dir.Name);
                Console.WriteLine("Parent: {0}", dir.Parent);
                Console.WriteLine("Creation: {0}", dir.CreationTime);
                Console.WriteLine("Attributes: {0}", dir.Attributes.ToString());
                Console.WriteLine("Root: {0}", dir.Root);
                Console WriteLine("**********************\n");
        }
}
```

То, что получилось при запуске этой программы на моем компьютере, представлено на рис. 11.3.

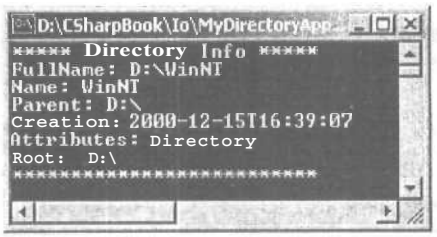


Рис. 11.3. Информация о каталоге D:\WinNT

Перечисление FileAttributes

Как мы уже могли увидеть в предыдущем примере, свойство Attributes позволяет получить информацию об атрибутах объекта файловой системы. Для этого свойства используются значения из перечисления FileAttributes (табл. 11.4).

Таблица 11.4. Некоторые значения перечисления FileAttributes

Значение	Описание
Archive	Этот атрибут используется приложениями при проведении резервного копирования, *а в* некоторых случаях — удаления старых файлов
Compressed	Определяет, что файл является сжатым
Directory	Определяет, что объект файловой системы является каталогом
Encrypted	Определяет, что файл является зашифрованным
Hidden	Определяет, что файл является скрытым (такой файл не будет выводиться при обычном просмотре каталога)
Normal	Определяет, что файл находится в обычном состоянии и для него установлены любые другие атрибуты, Этот атрибут не может использоваться с другими атрибутами
Offline	Файл (расположенный на сервере) кэширован в хранилище off-line на клиентском компьютере. Возможно, что данные этого файла уже устарели
Readonly	Файл доступен только для чтения
System	Файл является системным (то есть файл является частью операционной системы или используется исключительно операционной системой)

Получение доступа к файлам через объект DirectoryInfo

Через DirectoryInfo можно не только получать доступ к информации о каталоге, но и работать с файлами в нашем каталоге. Давайте расширим возможности нашего класса MyDirectory таким образом, чтобы он считывал информацию о всех файлах *.bmp, расположенных в каталоге D:\WinNT при помощи метода GetFiles(). Этот метод возвращает массив объектов FileInfo. Мы пройдем по всем элементам этого массива при помощи конструкции foreach и выведем о них информацию на консоль:

```
class MyDirectory
{
        public static void Main(String[] args)
        {
                // Создаем объект DirectoryInfo, соответствующий D:\WinNT
                DirectoryInfo dir - new DirectoryInfo(@"D:\WinNT");
                ...

                // Получаем все файлы с расширением BMP
                FileInfo[] bitmapFiles = dir.GetFiles("*.bmp");

                // А сколько их у нас?
                Console.WriteLine("Found {0} *.bmp files\n", bitmapFiles.Length);

                // Теперь выводим информацию о каждом файле
                foreach (FileInfo f in bitmapFiles)
                {
                        Console.WriteLine("*******************\n");
                        Console.WriteLine("File name: {0}", f.Name);
                        Console.WriteLine("File size: {0}", f.Length);
                        Console.WriteLine("Creation: {0}", f.CreationTime);
                        Console.WriteLine("Attributes: {0}", f.Attributes.ToString());
```

```
                Console.WriteLine("*******************\n");
            }
        }
}  .
```

То, что получилось на моем компьютере, представлено на рис. 11.4.

Рис. 11.4. Информация о файлах BMP в каталоге D:\WinNT

Создаем подкаталоги при помощи класса DirectoryInfo

Метод DirectoryInfo.CreateSubdirectory() позволяет создавать в выбранном нами каталоге как единственный подкаталог, так и множество подкаталогов (в том числе и вложенных друг в друга). Давайте создадим в каталоге D:\WinNT несколько дополнительных подкаталогов:

```
class MyDirectory
{
        public static void Main(String[] args)
        {
                DirectoryInfo dir = new DirectoryInfo(@"D:\WinNT");
                ...

                // Создаем в D:\WinNT новые подкаталоги
                try
                {
                        // Создаем D:\WinNT\MyFoo
                        dir.CreateSubdirectory("MyFoo");

                        // Создаем D:\WinNT\MyBar\MyQaaz
                        dir.CreateSubdirectory(@"MyBar\MyQaaz");
```

```
            }
            catch(IOException e) { Console.Write(e.Message);}
        }
}
```

После запуска этой программы в каталоге D:\WinNT появятся новые подкаталоги (рис. 11.5).

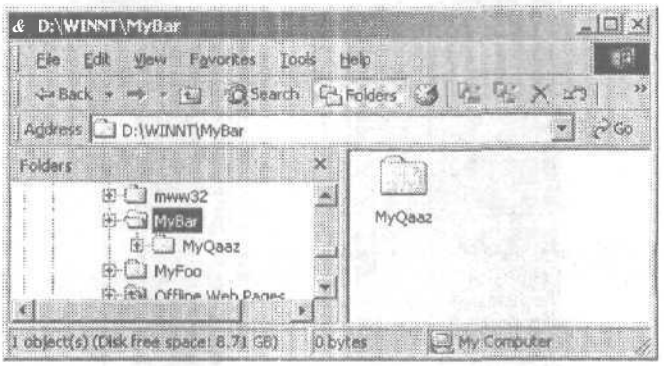


Рис. 11.5. Созданные при помощи CreateSubdirectory() подкаталоги

Метод CreateSubdirectory()**при успешном** завершении **возвращает объект** Directory, **которому соответствует возвращаемый** нами **подкаталог. В принципе, что-нибудь делать с этим возвращаемым объектом совсем не обязательно, но иногда это может оказаться полезным:**

```
try
{
        Directory d - dir.CreateSubdirectory("MyFoo");
        Console.WriteLine("Created: {0}", d.FullName);

        d = dir.CreateSubdirectory(@"MyBar\MyQaaz");
        Console.WriteLine("Created: {0}", d.FullName);
}
catchdOException e) { Console.WriteLine(e.Message); }
```

Статические члены класса Directory

Работать с каталогами в .NET **можно не только при помощи класса** DirectoryInfo, **с которым мы только что познакомились, но и при помощи класса** Directory. **Возможности этого класса, реализованные в виде статических членов, во** многом **совпадают с возможностями** DirectoryInfo (для **использования которых необходимо создавать отдельный объект), однако в** Directory предусмотрены **и уникальные** члены, аналогов которым в DirectoryInfo нет (например, метод GetLogicalDrives()). **Перечислять все члены** Directory мы не будем, **а просто проиллюстрируем его использование, добавив в наш класс** MyDirectory **новые возможности. Окончательный вариант** MyDirectory **будет перечислять буквы всех** дисков для данного компьютера, **а также использовать статический** метод Directory.Delete() для удаления **с компьютера только что созданных** каталогов \MyFoo **и** \MyBar\MyQaaz:

```
class MyDirectory
{
        public static void Main(String[] args)
        {
                // Создаем объект DirectoryInfo, соответствующий D:\WinNT
                DirectoryInfo dir = new DirectoryInfo(@"D:\WinNT");
                ...

                // А теперь воспользуемся несколькими статическими методами класса
                // Directory

                // Выводим информацию обо всех логических дисках
                string[] drives = Directory.GetLogicalDrives();
                Console.WriteLine("Here are your drives:");
                foreach(string s in drives)
                {
                        Console.WriteLine("->{0}", s);
                }

                // Удаляем только что созданные каталоги
                Console.Write(Going to delete\n->" + dir.FullName +
                        "\\MyBar\MyQaaz.\nand\an->" + dir.FullName +
                         \\MyFoo.\n" + "Press a key to continue!");
                Console.Read();

                try
                {
                        Directory.Delete(@"D:\WinNT\MyFoo");

                // Необязательный второй параметр определяет, будут ли удалены также
                // и все вложенные подкаталоги
                Directory.Delete(@"D:\WinNT\MyBar", true);
                }

                catch(IOException e)
                {
                        Console.WriteLine(e.Message);
                }
        }
}
```

Часть вывода нашего приложения, относящаяся к тому, что мы только что сейчас сделали, представлена на рис. 11.6.

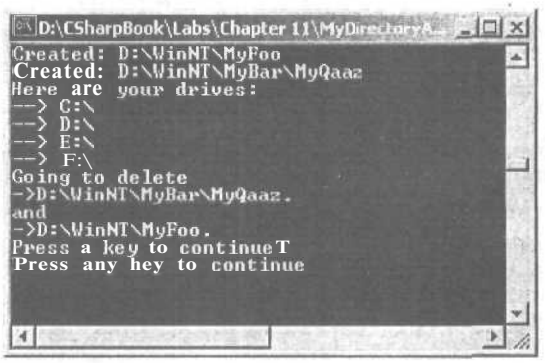


Рис. 11.6. Работаем со статическими членами Directory

К этому моменту мы уже умеем работать с каталогами — создавать их, удалять, получать информацию о самих каталогах и содержащихся в них файлах. Следующая наша задача — разобраться с тем, как производится работа с файлами в .NET.

Код приложения MyDirectoryApp можно найти в подкаталоге Chapter 11.

Класс FileInfo

Класс FileInfo представляет файл, содержащийся на жестком диске компьютера. Он позволяет получать информацию об этом файле (например, о времени его создания, размере, атрибутах и т. п.), а также производить различные операции, например по созданию файла или его удалению. Этот класс наследует множество членов от FileSystemInfo и, кроме того, обладает еще и набором собственных уникальных членов, которые представлены в табл. 11.5.

Таблица 11.5. Наиболее важные члены класса FileInfo

Член	Описание
AppendText()	Создает объект StreamWriter (о нем будет рассказано дальше) для добавления текста к файлу
CopyTo()	Копирует уже существующий файл в новый файл
Create()	Создает новый файл и возвращает объект FileStream (о нем также будет рассказано ниже) для взаимодействия с этим файлом
CreateText()	Создает объект StreamWriter для записи текстовых данных в новый файл
Delete()	Удаляет файл, которому соответствует объект FileInfo
Directory	Возвращает каталог, в котором расположен данный файл
DirectoryName	Возвращает полный путь к данному файлу в файловой системе
Length	Возвращает размер файла
MoveTo()	Перемещает файл в указанное пользователем место (этот метод позволяет одновременно переименовать данный файл)
Name	Позволяет получить имя файла
Open()	Открывает файл с указанными пользователем правами доступа на чтение, запись или совместное использование с другими пользователями
OpenRead()	Создает **объект** FileStream, доступный **только для чтения**
OpenText()	Создает объект StreamReader (о нем также будет рассказано ниже), который позволяет считывать информацию из существующего текстового файла
OpenWrite()	Создает объект FileStream, доступный для чтения и записи

Как мы видим, большинство методов FileInfo возвращает объекты (FileStream, StreamWriter, StreamReader и т. п.), которые позволяют различным образом взаимодействовать с файлом, например производить чтение или запись в него. Мы еще познакомимся с этими объектами, однако сейчас нам важно отметить, что при помощи класса FileInfo мы можем просто и очень удобно создать новый файл в файловой системе:

```
public class FileManipulator
{
        public static int Main(string[] args)
```

```
        {
            // Создаем новый файл в корневом каталоге диска C:
            FileInfo f = new FileInfo(@"C:\Test.txt");
            FileStream fs = f.Create();

            // Выводим основную информацию о созданной нами файле
            Console.WriteLine("Creation: {0}", f.CreationTime);
            Console.WriteLine("Full Name: {0}", f.FullName);
            Console.WriteLine("Full atts: {0}", f.Attributes.ToString());
            Console.Write("Press a key to delete file");
            Console.Read();

            // Закрываем FileStream и удаляем файл
            fs.Close();
            f.Delete();

            return 0;
        }
    }
```

Обратите внимание, что метод `Create()` возвращает объект `FileStream`, который позволяет взаимодействовать с файлом. После запуска нашего приложения мы сможем увидеть созданный файл (до его удаления) в корневом каталоге диска C: (рис. 11.7).

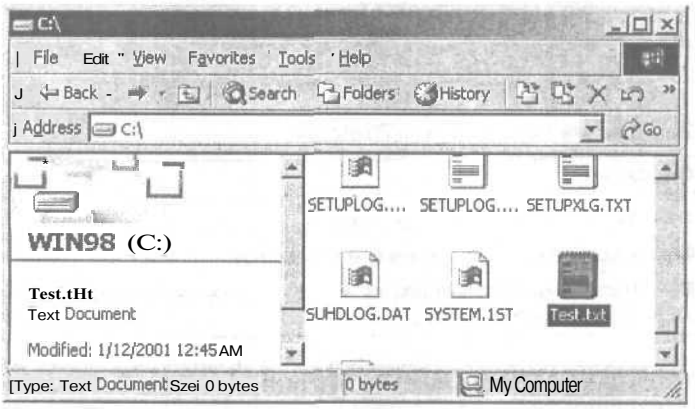


Рис. 11.7. Создаем файл на диске программным образом

Использование метода FileInfo.Open()

Метод `FileInfo.Open()` может быть использован как для открытия уже существующего файла, так и для создания нового с большим количеством возможностей, чем метод `FileInfo.Create()`. Вот пример использования этого метода:

```
// Открываем (или создаем) файл для чтения и записи, но без возможности совместного
// использования и сохраняем указатель на этот файл в объекте FileStream
FileInfo f2 = new FileInfo(@"C:\HelloThere.ini");
FileStream s = f2.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None);
s.Close();
f2.Delete();
```

Эта версия перегруженного метода Open() принимает три параметра. Первый из них определяет вид запроса на открытие файла (создание нового файла, открытие существующего файла, добавление к существующему файлу и т. п.), Для него используются значения из перечисления FileMode (табл. 11.6).

Таблица 11.6. Значения перечисления FileMode

Значение	Описание
Append	Открывает файл, если он существует, и ищет конец этого файла. Если указанный файл не существует, создается новый файл. Обратите внимание, что режим FileMode.Access может быть использован только совместно с доступом типа FileAccess.Write
Create	Указывает, что операционная система должна создать новый файл. Будьте осторожны — если в каталоге уже существует файл с таким же именем, он будет перезаписан!
CreateNew	Также определяет создание нового файла, но если файл с тем же именем уже существует в каталоге, будет сгенерировано исключение IOException
Open	Определяет, что операционная система должна открыть существующий файл
OpenOrCreate	Определяет, что операционная система должна открыть файл, если он существует. Если же нет, то файл с таким названием должен быть создан
Truncate	Определяет, что операционная система должна открыть существующий файл. После открытия он должен быть обрезан до нулевой длины

Второй параметр определяет тип доступа к файлу как к потоку байтов. Для него используются параметры из перечисления FileAccess (табл. 11.7).

Таблица 11.7. Значения перечисления FileAccess

Значение	Описание
Read	Файл будет открыт только для чтения
ReadWrite	Файл будет открыт и для чтения, и для записи
Write	Файл будет открыт только для записи (то есть данные будут добавляться в файл, но не считываться из него)

Третий параметр определяет возможности совместного доступа к открытому файлу. Набор значений для него — в перечислении FileShare (табл. 118).

Таблица 11.8. Значения перечисления FileShare

Значение	Описание
None	Совместное использование открытого файла запрещено. На любой запрос на открытие данного файла будет возвращено сообщение об ошибке
Read	Позволяет открывать файл одновременно и другим пользователям, но только на чтение. Если этот флаг не установлен, на любые запросы на открытие данного файла на чтение будет возвращаться сообщение об ошибке
ReadWrite	Позволяет открывать файл одновременно и другим пользователям на чтение и запись
Write	Позволяет открывать файл одновременно и другим пользователям на запись

Методы FileInfo.OpenRead() и FileInfo.OpenWrite()

Помимо метода Open(), в FileInfo предусмотрены также методы OpenRead() и OpenWrite(). Как вы, наверное, уже догадались, эти методы возвращают объекты FileStream, открытые только для чтения или только для записи. Например:

```
// Получаем объект FileStream с доступом только для чтения
FileInfo f3 = new FileInfo(@"C:\boot.ini");
FileStream readOnlyStream = f3.OpenRead();
readOnlyStream.Close();

// А теперь - объект FileStream с доступом только для записи
FileInfo fA = new FileInfo(@"C:\config.sys");
FileStream writeOnlyStream = f4.OpenWrite();
writeOnlyStream.Close();
```

Код приложения BasicFileApp можно найти в подкаталоге Chapter 11.

Методы FileInfo.OpenText(), FileInfo.CreateText() и FileInfo.AppendText()

Еще один метод класса FileInfo, позволяющий открывать файл для чтения, — метод OpenText(). Его главное отличие от Open(), OpenRead() и OpenWrite() заключается в том, что этот метод возвращает вместо FileStream объект StreamReader (подробнее о нем будет рассказано ниже):

```
// Получаем объект StreamReader
FileInfo f5 = new FileInfo(@"C:\bootlog.txt");
StreamReader sreader = f5.OpenText();
sreader.Close();
```

Последние два метода, которые представляют для нас интерес, — метод CreateText() и AppendText(). Оба эти метода возвращают объект StreamWriter:

```
// Получаем объекты StreamWriter
FileInfo f6 = new FileInfo(@"D:\AnotherTest.txt");
f6.Open(FileMode.Create, FileAccess.ReadWrite);
StreamWriter swriter = f6.CreateText();
swriter.Close();

FileInfo f7 = new FileInfo(@"D:\FinalTest.txt");
f7.Open(FileMode.Create, FileAccess.ReadWrite);
StreamWriter swriterAppend = f7.AppendText();
swriterAppend.Close();
```

К этому моменту мы с вами уже научились получать при помощи FileInfo объекты FileStream, StreamReader и StreamWriter. Что делать с этими объектами — мы с вами скоро узнаем. А пока отметим, что класс File позволяет делать все то же самое, что и FileInfo, но уже при помощи набора статических членов. Специально рассматривать класс File мы не будем, а вместо этого просто будем использовать класс File в наших примерах. Если же вам потребуется дополнительная информация по классу File, то ее легко можно найти в электронной справке по Visual Studio.NET.

Абстрактный класс Stream

В терминах реализации ввода-вывода в программах stream (поток) — это сущность, предназначенная для работы с блоками данных. Абстрактный класс System.IO.Stream

определяет значительное число членов, которые обеспечивают как синхронное, так и асинхронное взаимодействие со средой хранения данных (файлом на диске или областью в оперативной памяти). Иерархия классов, производных от System.IO.Stream, представлена на рис. 11.8.

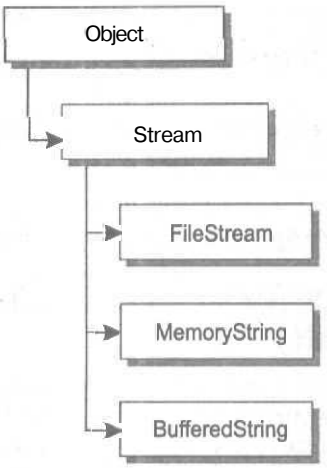


Рис. 11.8. Классы, производные от Stream

Все классы, производные от Stream, предназначены для работы с блоками двоичных данных (для текстовых данных есть свои классы). Кроме того, классы, производные от Stream, поддерживают поиск в потоке данных — то есть обнаружение местонахождения последовательности двоичных символов в потоке. Наиболее важные члены базового класса Stream представлены в табл. 11.9.

Таблица 11.9. Члены класса Stream

Член	Описание
CanRead CanSeek CanWrite	Определяют, будет ли данный поток поддерживать чтение, поиск и (или) запись
Close()	Закрывает текущий поток и освобождает связанные с ним ресурсы (сокеты, указатели на файлы и т. п.)
Flush()	Записывает данные из буфера в связанный с потоком источник данных и очищает буфер. Если для данного потока буфер не используется, то этот метод ничего не делает
Length	Возвращает длину потока в байтах
Position	Определяет указатель на местонахождение (позицию) в текущем потоке
Read() ReadByte()	Считывают последовательность байтов (или единственный байт) в текущем потоке и перемещают указатель в потоке на количество считанных байтов
Seek()	Устанавливает указатель на местонахождение (позицию) в текущем потоке
SetLength()	Устанавливает длину текущего потока
Write() WriteByte()	Записывают последовательность байтов (или единственный байт) в текущий поток и перемещают указатель в потоке на количество записанных байтов

Работа с объектом FileStream

Класс FileStream обеспечивает реализацию абстрактных членов класса Stream для работы с файлами на диске. Так же как DirectoryInfo и FileInfo, FileStream обеспечивает возможность открытия существующих файлов и создания новых. Как правило, при создании объектов FileStream используются значения из перечислений FileMode, FileAccess и FileShare. Например, при помощи следующего кода мы можем создать файл test.dat в текущем каталоге приложения:

```
// Создаем файл в текущем каталоге
FileStream myFStream = new FileStream("test.dat", FileMode.OpenOrCreate,
                                                  FileAccess.ReadWrite);
```

Поэкспериментируем с возможностями чтения и записи в файл при помощи объекта FileStream. Для записи потока байтов в файл используются методы WriteByte() и Write(), при применении которых внутренний указатель на место в файле перемещается автоматически. Для чтения байтов из файла можно воспользоваться методами Read() или ReadByte(). Вот пример кода:

```
// Записываем байты в файл *.dat
for(int i = 0; i < 256; i++)
{
        myFStream.WriteByte((byte)i);
}

// Переставляем внутренний указатель на начало
myFStream.Position = 0;

// Считываем байты из файла *.dat
for(int i = 0; i < 256; i++)
{
        Console.Write(myFStream.ReadByte());
}
myFStream.Close();
```

Если мы откроем созданный нами файл в Visual Studio.NET, то увидим генерированный нами поток байтов (рис. 11.9).

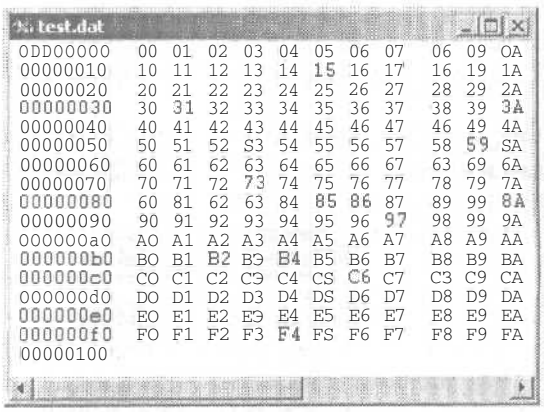


Рис. 11.9. Двоичный дамп созданного нами файла

Класс MemoryStream

Работа с классом MemoryStream во многом напоминает работу с FileStream с теми отличиями, которые возникают из-за того, что мы записываем поток двоичных данных не в файл на диске, а в оперативную память. И FileStream, и MemoryStream происходят от одного базового класса — Stream, и поэтому многие их члены являются общими. Мы можем обновить код нашего предыдущего примера для работы с MemoryStream следующим образом:

```
// Создаем объект MemoryStream точно определенного объема
MemoryStream myMemStream = new MemoryStream();
myMemStream.Capacity = 256;

// Записываем байты в myMemStream
for(int i = 0; i < 256; i++)
{
        myMemStream.WriteByte((byte)i);
}

// Переставляем внутренний указатель на начало
myMemStream.Position = 0;

// Считываем байты из потока
for(int i = 0; i < 256; i++)
{
        Console.Write(myMemStream.ReadByte());
}
myMemStream.Close();
```

Результат работы этой программы будет аналогичен результату из предыдущего примера. Единственная разница заключается в том, куда мы записываем и откуда считываем информацию — из файла или оперативной памяти. Помимо членов, унаследованных от Stream, MemoryStream определяет еще несколько интересных членов (с одним из них — Capacity мы уже столкнулись в нашем примере). Эти члены представлены в табл. 11.10.

Таблица 11.10. Наиболее важные члены MemoryStream

Член	Описание
Capacity	**Позволяет получить или установить количество** байтов, **выделенных под этот поток**
GetBuffer()	**Возвращает массив байтов, при помощи которых поток был создан**
ToArray()	**Записывает все** содержимое **потока в массив байтов, вне зависимости от свойства Position**
WriteTo()	**Записывает** все содержимое данного **объекта MemoryStream в** другой **объект** класса, **производного от** Stream (например, **в FileStream)**

Обратите внимание, что разработчики библиотеки базовых классов предусмотрели возможность взаимодействия между MemoryStream и FileStream. Например, при помощи метода WriteTo() мы можем без каких-либо проблем передать данные из оперативной памяти в файл. Кроме того, мы можем переместить все данные из MemoryStream в массив байтов:

```
// Сбрасываем данные из MemoryStream в файл
FileStream dumpFile = new FileStream("Dump.dat", FleMode.Create, FileAccess.ReadWrite);
myMemStream.WriteTo(dumpFile);

// Сбрасываем данные из MemoryStream в массив байтов
byte[] bytesinMemory = myMemStream.ToArray();
myMemStream.Close();
```

Класс BufferedStream

Последний класс, производный от Stream, который мы рассмотрим, — класс BufferedStream. Объект этого класса может быть использован как временное хранилище; для информации, которая затем будет передана в постоянное хранилище. Например, предположим, что мы открыли файл и собираемся записать в него несколько блоков двоичных данных. В принципе, каждый из этих блоков можно записывать отдельно при помощи FileStream.Write(), но если нам важна производительность, то лучше будет вначале записать все блоки в объект BufferedStream, а затем уже перенести все разом в файл на диске. Этим способом мы уменьшим число обращений к физическому файлу на диске. Выглядеть все это в коде программы может так:

```
// Создаем объект BufferedStream. подключенный к уже имеющемуся объекту FileStream
BufferedStream myFileBuffer = new BufferedStream(dumpFile);

// Добавляем в BufferedStream несколько байтов
byte[] str = {127, 0x77. 0x4. 0x0. 0x0. 0x16};
myFileBuffer.Write(str, 0, str.Length);

// А теперь записываем все содержимое BufferedStream в файл
myFileBuffer.Close();
```

Код приложения Streamer, в котором приведены примеры работы с классами FileStream, MemoryStream и BufferedStream, можно найти в подкаталоге Chapter *i* 1.

Классы StreamWriter и StreamReader

Классы StreamReader и StreamWriter пригодятся нам в тех ситуациях, когда необходимо считать или записать символьные данные (данные в формате string). По умолчанию оба этих типа работают с кодировкой Unicode. Если нас это по какой-то причине не устраивает, мы можем изменить используемую кодировку при помощи объекта System.Text.Encoding. Для простоты мы будем считать, что во всех наших примерах Unicode нас вполне устраивает (однако я советую вам познакомиться с пространством имен System.Text — вы найдете там немало интересных возможностей).

Класс StreamReader производится от абстрактного класса TextReader, так же как и класс StringReader (разговор о нем нам еще предстоит). Базовый класс TextReader обеспечивает производным классам крайне ограниченный набор возможностей, в основном, конечно же, связанных с чтением символьных данных.

Для класса StreamWriter (так же как и для класса StringWriter — о нем будет сказано позже) базовым является класс TextWriter. Этот класс определяет возможности для записи символьных данных в поток. Отношения между классами System. IG для работы с символьными данными представлены на рис. 11.10.

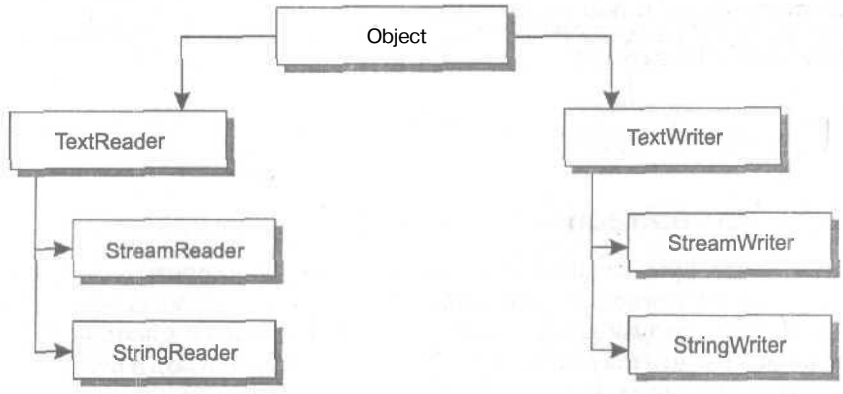


Рис. 11.10. Иерархия классов для StreamReader и StreamWriter

Значительная часть функциональных возможностей класса StreamWriter обеспечивается при помощи членов, унаследованных от TextWriter. Эти члены представлены в табл. 11.11.

Таблица 11.11. Наиболее важные члены базового класса TextWriter

Член	Описание
Close()	Закрывает соответствующий объект Writer и освобождает связанные с ним ресурсы. Если в процессе записи используется буфер, он будет автоматически очищен
Flush()	Очищает все буферы для текущего объекта Writer и записывает накопленные в них данные в место постоянного их хранения, но при этом сам объект Writer не закрывается
NewLine	Используется для определения последовательности символов, означающих начало новой строки. По умолчанию используется последовательность «возврат каретки» — «перевод строки» (\r\n)
Write()	Записывает новый отрезок текста в поток без применения последовательности начала новой строки
WriteLine()	Записывает новую строку в поток (с применением последовательности начала новой строки)

Последние два члена TextWriter из табл. 11.11 выглядят как-то очень знакомо. Действительно, Write() и WriteLine() в TextWriter делают то же самое, что и в System.Console — единственное отличие заключается в том, что отрезок текста или строка выводится не на системную консоль, а в поток (например, в файл).

Производный класс StreamWriter обеспечивает реализацию абстрактных методов Write(), Close() и Flush(), а также определяет дополнительное свойство AutoFlush. Если это свойства имеет значение true, то буферы будут очищаться после каждого выполнения операции записи. Конечно, этой возможностью следует пользоваться только в особых случаях, поскольку с точки зрения производительности гораздо эффективнее будет установить для AutoFlush значение false.

Запись в текстовый файл

Проиллюстрируем работу со StreamWriter на примере. Класс, который мы создадим, будет создавать файл с именем thoughts.txt при помощи класса FileInfo. Да-

лее мы используем метод CreateText() для получения объекта StreamWriter. После этого мы уже сможем добавлять в файл нужные нам строки текста:

```
public class MyStreamWriterReader
{
        public static int Main(string[] args)
        {
                // Создаем файл
                FileInfo f = new FileInfo("Thoughts.txt");

                // Получаем объект StreamWriter и с его помощью записываем в файл
                // несколько строк текста
                StreamWriter writer = f.CreateText();
                writer.WriteLine("Don't forget Mother's Day this year...");
                writer.WriteLine("Don't forget Father's Day this year...");
                writer.WriteLine("Don't forget these numbers:");

                for(int i = 0; i < 10; i++);
                t
                        writer.Write(i + " ");
                )
                // Вставляем символ начала новой строки
                writer.Write(writer.NewLine);

                // Метод Close() автоматически очищает все буферы!
                writer.Close();
                Console.WriteLine("Created file and wrote some thoughts...");
        }
I
```

Если мы откроем созданный нами файл в Notepad, то сможем увидеть созданное нами его текстовое содержимое (рис. 11.11).

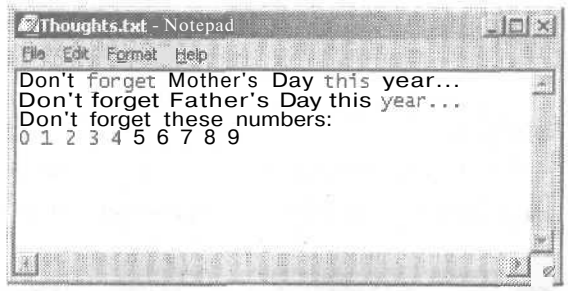


Рис. 11.11. Содержимое файла thoughts.txt

Все очень просто. Отметим только, что методы Write() и WriteLine() многократно перегружены, чтобы обеспечить нам самые разные возможности добавления символьных и числовых данных.

Считывание информации из текстового файла

Записывать текстовую информацию в файл мы уже умеем. Осталось научиться такую информацию считывать. Как вы, наверное, уже догадались, проще всего это сделать при помощи класса StreamReader. Члены StreamReader, унаследованные от базового класса TextReader, представлены в табл. 11.12.

Таблица 11.12. Наиболее важные члены класса TextReader

Член	Описание
Peek()	Возвращает следующий символ, не изменяя позицию указателя в файле
Read()	Считывает данные из потока на входе
ReadBlock()	Считывает указанное пользователем количество символов, начиная с определенной позиции, и записывает считанные данные в буфер
ReadLine()	Считывает строку данных из текущего потока и возвращает ее как значение типа string. Пустая строка (null string) означает конец файла (EOF)
ReadToEnd()	Считывает все символы, начиная с текущей позиции и до конца потока, и возвращает считанные данные как единое значение типа string

Давайте расширим возможности нашего класса MyStreamWriterReader таким образом, чтобы он мог считывать данные при помощи StreamReader из только что созданного нами текстового файла и выводить эти данные на консоль:

```
public class MyStreamWriterReader
{
        public static int Main(string[] args)
        {
                // Код для записи файла остается прежним
                ...

                // А теперь выводим информацию из файла на консоль при помощи
                // StreamReader
                Console.WriteLine("Here ane your thoughts:\n");
                StreamReader sr = File.OpenText("Thoughts.txt");

                stning input = null;
                while ((input = sr.ReadLine()) !=null)
                {
                        Console.WriteLine(input);
                }
                sr.Close();

                return 0;
        }
}
```

Результат работы этой программы представлен на рис. 11.12.

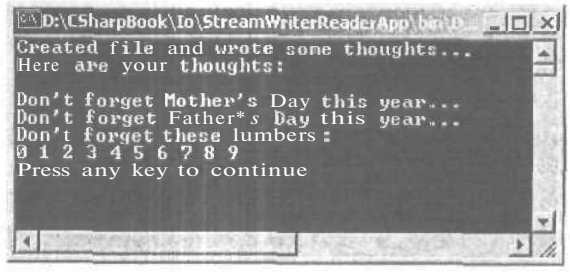


Рис. 11.12. Считываем данные из файла

В нашем примере мы получаем объект StreamReader при помощи метода File.OpenText(). Далее мы считываем все строки и выводим их на консоль до того

момента, пока нам не попадется пустая строка (означающая конец файла). Однако все можно сделать проще, если воспользоваться методом ReadToEnd():

```
// Давайте сразу все!
string allOfTheData = sr.ReadToEnd();
MessageBox.Show(allOfTheData, "Here it is:");
sr.Close();
```

Таким образом, если нам потребовалось записать какую-либо информацию в текстовый файл или считать информацию из текстового файла — классы StreamReader и StreamWriter в нашем распоряжении.

Код приложения StreamWriterReaderAppможно найти в подкаталоге Chapter 11.

Класс StringWriter

При помощи классов StringWriter и StringReader мы можем обращаться к текстовой информации как к потоку в оперативной памяти. Такой подход может пригодиться, если мы добавляем текстовую информацию в специальный буфер в оперативной памяти. Для получения доступа к этому буферу мы можем воспользоваться тем же экземпляром класса StringWriter, вызвав для него замещенный метод ToString() (этот метод вернет объект класса System.String). Мы можем также воспользоваться методом StringWriter.GetStringBuilder(). Как несложно догадаться, этот метод возвращает объект System.Text.StringBuilder, пользоваться которым часто бывает удобнее (подробнее о нем было рассказано в главе 2).

Давайте переделаем наш предыдущий пример так, чтобы информация сбрасывалась в объект StringWriter вместо файла на диске. Поскольку возможности StringWriter и StreamWriter очень схожи, программы будут почти неотличимы;

```
public class MyStreamWriterReader
{
    public static int Main(string[] args)
    {
        // Получаем обьект StringWriter и с его помощью записываем
        // в файл несколько строк текста
        StringWriter writer = new StringWriter();
        writer.WriteLine("Don't forget Mother's Day this year...");
        writer.WriteLine("Don't forget Father's Day this year...");
        writer.WriteLine("Don't forget these numbers:");

        for(int i - 0; i < 10: i++);
        {
            writer.Write(i + " ");
        }
        // Вставляем символ начала новой строки
        writer.Write(writer.NewLine);

        // Метод Close() автоматически очищает все буферы!
        writer.Close();
        Console.WriteLine("Stored thoughts in a StringWriter...");

        // Получаем копию содержимого StringBuffer (в виде значения типа string)
        // и выводим ее на консоль
        Console.WriteLine("Contents: {0}", writer.ToString());
```

```
            return 0;
        }
}
```

Результат работы этой нехитрой программы представлен на рис. 11.13.

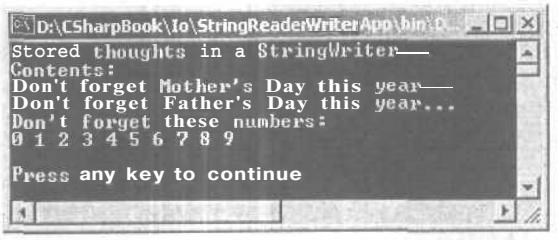


Рис. 11.13. Используем для хранения символьных данных объект StringWriter

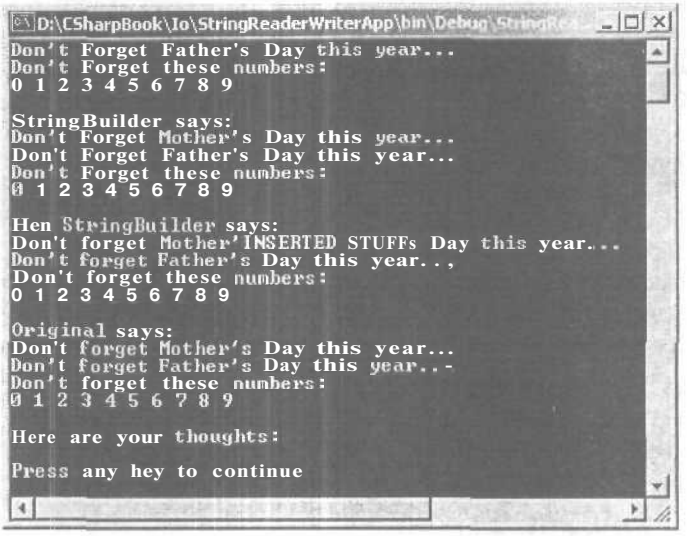


Рис. 11.14. Применение StringBuilder совместно со StringWriter

Теперь давайте получим доступ к содержимому StringWriter через объект StringBuilder:

```
// Это нам потребуется для StringBuilder
using System.Text;

public class MyStreamWriterReader
{
    public static int Main(string[] args)
    {
        // Здесь все остается как в предыдущем примере
        ...

        // Получаем объект StringBuilder и выводим его содержимое
        StringBuilder str = writer.GetStringBuilder();
        string allOfTheData = str.ToString();
```

```
            Console.WriteLine("StringBuilder says:\n{0} ", allOfTheData);

            // Вставляем в буфер новый элемент, позиция вставки 20
            str.Insert(20, "INSERTED STUFF");
            allOfTheData = str.ToString();
            Console.WriteLine("New StringBuilder says:\n{0}", allOfTheData);

            // Удаляем вставленный элемент
            str.Remove(20, "INSERTED STUFF".Length);
            allOfTheData • str.ToStringO;
            Console.WriteLine("Original says:\n{0}", allOfTheData);

            return 0;
        }
}
```

Таким образом, при помощи StringBuilder можно не только получить доступ к содержимому буфера с текстом, но и произвести с ним нужные нам операции. В этом легко убедиться, посмотрев на рис. 11.14.

Класс StringReader

Класс Strl ngReader, как вы уже, наверное, догадываетесь, нужен для считывания символьной информации из буфера в оперативной памяти точно так же, как это делает класс StreamReader для файла. Его применение выглядит следующим образом:

```
// Считываем информацию при помощи StringReader
StringReader sr = new StringReader(writer.ToString());

string input = null;
while ((input = sr.ReadLine()) != null)
{
        Console.WriteLine(input);
}
sr.Close;
```

Единственное, что осталось упомянуть, — то, что у классов, производных от TextReader и TextWriter, есть одно серьезное ограничение: с их помощью нельзя обеспечить произвольный доступ к содержимому блока символов. Например, в Stri ngReader нет таких членов, которые позволили бы нам перемещать указатель внутри содержимого буфера или «перепрыгивать» через какое-то количество символов и начинать чтение заданной точки. Для произвольного доступа необходимо использовать какой-либо из классов, производных от Stream.

Код приложения StringReaderWriterApp можно найти в подкаталоге Chapter 11.

Работа с двоичными данными (классы BinaryReader и BinaryWriter)

Последние два важнейших класса пространства имен System. IO, которые нам осталось рассмотреть, — это классы BinaryReader и BinaryWriter. Оба этих класса происходят непосредственно от System.Object, как показано на рис. 11.15.

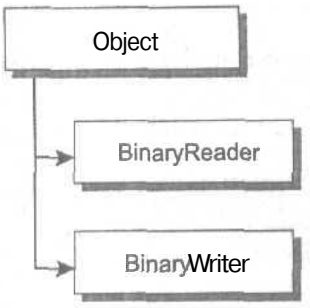


Рис. 11.15. Происхождение классов BinaryReader и BinaryWriter

Эти типы позволяют считывать и записывать определенные двоичные типы данных в поток. Класс BinaryWriter определяет многократно перегруженный метод Write() для помещения в поток объектов самых разных типов данных. Кроме того, в этом классе также определено еще несколько членов, которые выглядят очень знакомо (табл. 11.13).

Таблица 11.13. Наиболее важные члены класса BinaryWriter

Член	Описание
BaseStream	Представляет поток, с которым работает объект BinaryWriter
Close()	Закрывает поток
Flush()	Очищает буфер
Seek()	Устанавливает позицию в текущем потоке
Write()	Записывает значение в текущий поток

Наиболее важные члены классы BinaryReader представлены в табл. 11.14.

Таблица 11.14. Наиболее важные члены класса BinaryReader

Член	Описание
BaseStream	Представляет поток, с которым работает объект BinaryReader
Close()	Закрывает объект BinaryReader
PeekChar()	Возвращает следующий символ без перемещения внутреннего указателя в потоке
Read()	Считывает поток байтов или символов и сохраняет в массиве (передаваемом как входящий параметр)
ReadXXXX()	Считывает данные определенного типа из потока (например, ReadBoolean(), ReadByte(), ReadInt32() и т. д.)

Проиллюстрируем возможности BinaryWriter и BinaryReader на примере. Класс ByteTweaker создает новый файл *.dat при помощи объекта FileStream и записывает в него символьные данные. После этого объект FileStream передается конструктору класса BinaryWriter (конструктор BinaryWriter может принимать в качестве входящего параметра объекты любых классов, производных от Stream, например, FileStream, MemoryStream или BufferedStream). После того как запись данных будет завершена, данные будут считаны при помощи объекта BinaryReader:

```
public class ByteTweaker
{
    public static int Main(string[] args)
    {
        Console.WriteLine("Creating a file and writing binary data...");
        FileStream myFStream = new FileStream("temp.dat", FileMode.OpenOrCreate,
                                              FileAccess.ReadWrite);

        // Записываем двоичные данные
        BinaryWriter binWrit = new BinaryWriter(myFStream);
        binWrit.WriteString("Hello as binary info...");
        int myInt = 99;
        float myFloat = 9984.82343F;
        bool myBool = false;
        char[] myCharArray = {'H', 'e', 'l', 'l', 'o'};
        binWrit.Write(myInt);
        binWrit.Write(myFloat);
        binWrit.Write(myBool);
        binWrit.Write(myCharArray);

        // Устанавливаем внутренний указатель на начало
        binWrit.BaseStream.Position = 0;

        // Считываем двоичную информацию как поток байтов
        Console.WriteLine("Reading binary data...");
        BinaryReader binRead = new BinaryReader(myFStream);
        int temp = 0;
        while(binRead.PeekChar() != -1)
        {
            Console.Write(binRead.ReadByte());
            temp = temp + 1;
            if(temp == 5)
            {
                // Добавляем пустую строку через каждые 5 байтов
                temp = 0;
                Console.WriteLine();
            }
        }

        // Все закрываем
        binWrit.Close();
        binRead.Close();
        myFStream.Close();
    }
}
```

Результат работы программы представлен на рис. 11.16.

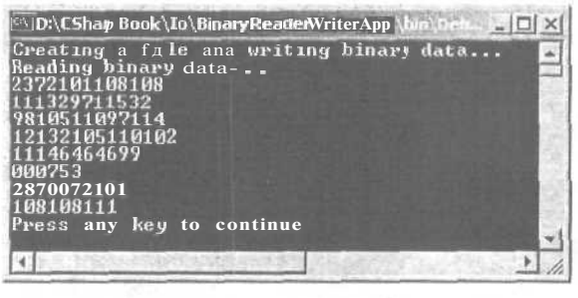


Рис. 11.16. Сеанс записи и чтения двоичных данных

Заметки на полях

Вполне возможно, что нам никогда и не потребуется выводить отдельные байты в поток. Однако все-таки заметим, что многие типы из самых разных пространств имен .NET используют знакомые нам классы System.IO в скрытом виде. Например, в классе System.Windows.Forms.Bitmap определен метод Save(), который записывает созданное нами двоичное изображение в файл. Вполне возможно создать новый объект Bitmap, передав в качестве входящего параметра объекта класса, производного от Stream. Таким образом мы, к примеру, можем изменять информацию о пикселах в объекте Bitmap в процессе выполнения программы. В принципе можно сосчитать координаты X и Y вручную, но проще использовать метод SetPixel(), например, так:

```
// Открываем файл изображения в каталоге приложения
Console.WriteLine("Modifying a bitmap in memory");
myFStream - new FileStream("Paint Splatter.bmp", FileMode.Open, FileAccess.ReadWrite);

// Создаем объект Bitmap на основе открытого потока
Bitmap rawBitmap = new Bitmap(myFStream);

// Рисуем белый крест поперек изображения (наш код применим лишь в том случае,
// если высота и ширина изображения одинаковы)
for(int i = 0; i < rawBitmap.Width; 1++)
{
        rawBitmap.SetPixel(i, 1, Color.White);
        rawBitmap.SetPixel(rawBitmap.Width - i) - 1, i - 1. Color.White);
}

// А теперь сохраняем измененное изображение в файл
rawBitmap .Save("newImage.bmp");
myFStream.Close();
```

На рис. 11.17 показано исходное изображение (файл splatter.bmp).

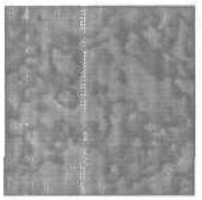


Рис. 11.17. Исходное изображение

А на рис. 11.18 — изображение после того, как с ним поработала наша программа.

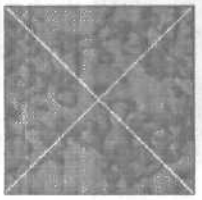


Рис. 11.18. Измененное изображение

На этом обзор наиболее важных типов пространства имен System. IO завершен. К этому моменту мы уже умеем записывать текстовую и двоичную информацию на диск и в оперативную память и считывать ее оттуда. Далее в этой главе будет рассказано о том, как в .NET реализована поддержка сериализации пользовательских типов.

Код приложения BinaryReaderWriter можно найти в подкаталоге Chapter 11.

Сохранение объектов в .NET

Как мы уже могли убедиться в первой части главы, в пространстве имен System. IO предусмотрено множество типов, которые позволяют записывать двоичные и символьные данные в какое-то место для хранения (например, файл на диске или буфер в памяти) и считывать их. Однако до сих пор нам не встретились типы, которые позволили бы сохранять объекты классов целиком и затем восстанавливать их. Об этом процессе, который называется сериализацией, и пойдет речь в оставшейся части главы.

В терминах .NET сериализация (serialization) — это термин, описывающий процесс преобразования объекта в линейную последовательность байтов. Обратный процесс, когда из потока байтов, содержащего всю необходимую информацию, объект восстанавливается в исходном виде, называется десериализацией (deserialization). Службы сериализации в .NET — это весьма сложные программные модули, Они обеспечивают многие неочевидные вещи: например, когда объект *сериализуется* в поток, информация о всех других объектах, на которые он ссылается, также должна сериализоваться. Например, когда сериализуется производный класс, ссылки на другие классы, которые есть в базовых классах для этого производного класса, также должны отслеживаться и учитываться.

После того как набор объектов сохранен в поток, мы можем обходиться с полученным набором байтов так, как нам захочется. Например, предположим, что мы сериализовали набор объектов в MemoryStream, то есть в буфер в оперативной памяти. Далее эта информация может быть передана на удаленный компьютер, в буфер обмена Windows, на CD или просто в файл на диске. Самим байтам абсолютно безразлично, где они будут лежать. Главное — чтобы они точно отображали весь набор подвергшихся сериализации объектов.

Графы для отношений объектов

Набор взаимосвязанных объектов, сериализованных в поток, называется графом объектов (object graph). Графы позволяют фиксировать отношения объектов друг к другу, и они *не соответствуют* классическим моделям отношений классов в объектно-ориентированном программировании. Внутри графа каждому из объектов присваивается уникальный номер, который используется только для служебных целей самого графа и которому совершенно не обязательно должно что-то соответствовать в реальном мире. Далее записывается информация о соответствии имени класса этому номеру, информация о всех отношениях этого класса с другими классами и отношениях других классов между собой. На самом деле все очень просто, достаточно один раз рассмотреть это на примере.

Разбираться с графом мы будем на примере, как обычно, из мира автомобилей. Пусть у нас будет базовый класс Car, который при помощи отношения «has-a» включает в себя класс Radio. Производный от класса Car класс JamesBondCar расширяет возможности базового класса. Объектный граф, моделирующий отношения между тремя этими классами, представлен на рис. 11.19.

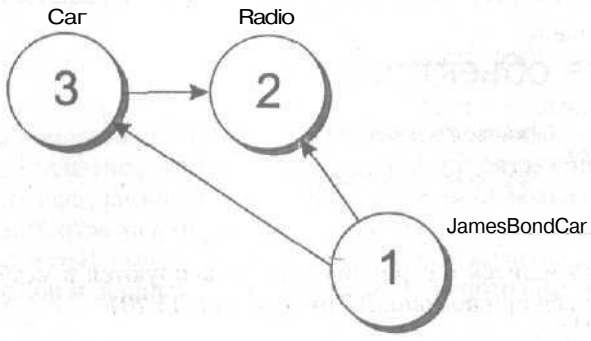


Рис. 11.19. Простой объектный граф

На рис. 11.19 ясно видно, что класс Car связан с классом Radio (при помощи отношения «has-a». Класс JamesBondCar также неразрывно связан с классом Car — как производный класс. Конечно же, JamesBondCar связан и с классом Radio, поскольку наследует его от Car. Учитывая, что каждому из классов присвоен уникальный номер, как показано на рисунке, формула графа может выглядеть следующим образом:

```
[Car 3. ref 2]. [Radio 2], [JamesBondCar 1, ref 3. ref 2]
```

Эта формула будет помещена при сериализации в место постоянного хранения вместе со значениями всех переменных классов Car, Radio и JamesBondCar. Поскольку объект Car (номер 3) связан с объектом Radio (номер 2), а объект JamesBondCar (номер 1) — с объектами 2 и 3, при сериализации в поток объекта JamesBondCar в этом процессе должны обязательно принимать участие и объект Car, и объект Radio. Пожалуй, самое замечательное, что можно сказать о создании объектных графов в процессе сериализации в .NET, — то, что их создание производится полностью автоматически и не требует никакого участия со стороны программиста.

Настройка объектов для сериализации

Чтобы можно было провести сериализацию объекта, каждый класс, который будет участвовать в сериализации, должен обладать атрибутом [Serializable]. Вот и все (и это правда). Если же мы решим, что какие-либо переменные данного класса должны быть исключены из сериализации, достаточно просто пометить их как [NonSerialized]. Обычно так помечаются те данные класса, которые «запоминать» не нужно (к примеру, те, которые на самом деле являются константами). Например, вот класс Radio, помеченный как доступный для сериализации, за исключением единственной переменной:

```
// Класс Radio может быть сериализован
[Serializable]
public class Radio
{
        // Однако нам нет необходимости сохранять это число
        [NonSerialized]
        private int objectIDNumber - 9:

        public Radio(){}
        public void On(bool state)
        {
                if(state - - true)
                        MessageBox.Show("Music is on...");
                else
                        MessageBox . Show("No tunes...");
        }
}
```

Атрибуты, относящиеся к сериализации, фиксируются в метаданных типов, в чем легко убедиться при помощи ILDasm.exe (рис. 1 1.20).

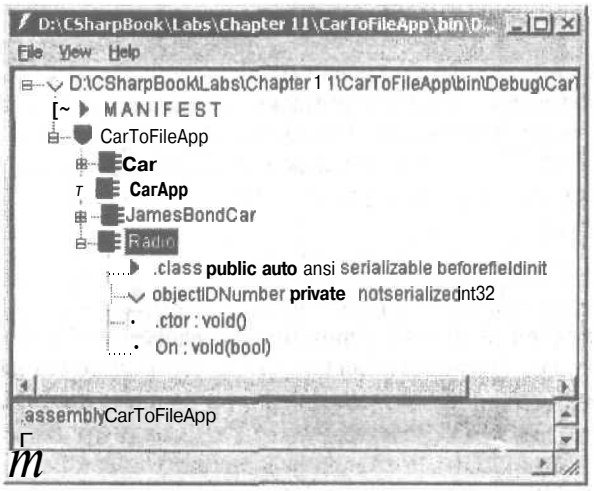


Рис. 11.20. Атрибуты Serializable и NonSerialized

Чтобы полностью подготовить наши автомобильные классы к сериализациям, которые предстоят нам в будущих разделах, дадим определения каждого из этих классов со всеми положенными атрибутами:

```
// Класс Car будет доступен для сериализации
[Serializable]
public class Car
t
        protected string petName;
        protected int maxSpeed;
        protected Radio theRadio - new Radio();

        public Car(string PetName. int maxSpeed)
        {
```

```
            this.petName - PetName;
            this.maxSpeed = maxSpeed;
        }
        // Пусть значения всех переменных будут автоматически установлены по умолчанию
        public Car() {}

        public string PetName
        {
            get { return petName; }
            set { petName - value; }
        }

        public int MaxSpeed
        {
            get { return maxSpeed; }
            set { maxSpeed - value; }
        }

        public void TurnQnRad1o(bool state)
        {
            theRadio.On(state);
        }
}

// Класс JamesBondCar будет также доступен для сериализации!
[Serializable]
public class JamesBondCar : Car
{
        protected bool isFlightWorthy;
        protected bool isSeaWorthy;

        public JamesBondCar(){}

        public JamesBondCar(string petName. 1nt maxSped, bool canFly, bool canSubmerge) :
                                                        base(petName, maxSpeed)
        {
            this.isFlightWorthy - canFly;
            this.isSeaWorthy - canSubmerge;
        }

        public void Fly()
        {
            . if(isFlightWorthy)
                    MessageBox.Show("Taking off!");
            else
                    MessageBox.Show("Falling off cliff!");
        }

        public void GoUnderWater()
        {
            if(isSeaWorthy)
                    MessageBox.Show("Diving..!");
            else
                    MessageBox.Show("Drowning!!!");
        }
}
```

Выбираем объект Formatter

После того как мы пометили класс как доступный для сериализации, наша следующая задача — выбрать формат, в котором будет сохранен объектный граф. Пространство имен System.Runtime.Serialization.Formatters включает в себя еще два пространства имен— *.Binary и *.Soap, каждому из которых соответствует один из двух объектов Formatter, которые можно использовать по умолчанию. Класс BinaryFormatter сериализует объектный граф в компактном потоке двоичного формата, в то время как класс SoapFormatter представляет граф как сообщение протокола SOAP (Simple Object Access Protocol — простого протокола доступа к объектам) в формате XML.

Класс BinaryFormatter определен в библиотеке mscorlib.dll, поэтому единственное, что нам потребуется для сериализации при помощи объекта Formatter, — определить использование этого пространства имен:

```
// Для сериализации объектов в двоичном формате
using System.Runtime.Serialization.Formatters.Binary;
```

Класс SoapFormatter определен в отдельной сборке, поэтому для сохранения объекта в формате SOAP вам вначале потребуется добавить ссылку на сборку System.Runtime.Serializaton.Formatters.Soap.dll, а затем использовать аналогичную команду:

```
// Для сериализации объектов в формате SOAP
using System.Runtime.Serialization.Formatters.Soap;
```

Пространство имен System.Runtime.Serialization

Если нас не устроит ни сериализация в двоичном формате, ни сериализация в формате SOAP, мы можем принять решение о создании своего собственного формата сериализации (и соответствующего ему объекта Formatter). Для этой цели нам пригодятся классы из пространства имен System.Runtime.Serialization. Кроме того, при настройке наших объектов для сериализации каким-либо специальным способом нам также пригодятся типы из этого пространства имен. Созданием своего собственного формата сериализации и объекта Formatter мы сейчас заниматься не будем, но типы, которые могут для этого пригодиться, все-таки приведем (табл. 11.15).

Таблица 11.15. Наиболее важные типы пространства имен System.Runtime.Serializaton

Тип	Описание
Formatter	Абстрактный базовый класс, который обеспечивает наиболее важные возможности для объектов Formatter в процессе сериализации
ObjectIDGenerator	Генерирует идентификаторы для объектов в объектном графе
ObjectManager	Управляет объектами в процессе десериализации
SerializationBuilder	Абстрактный базовый класс, обеспечивающий возможности по сериализации объекта в поток
SerializationInfo	Используется объектами со специальным «поведением» во время сериализации. Объект SerializationInfo объединяет в себе все данные, необходимые для сериализации или десериализации объекта. Можно сказать, что этот класс — нечто вроде хранилища пар имя — значение, представляющих внутреннее состояние объекта

Кроме этих типов, в пространстве имен System.Runtime.Serialization определены также два интерфейса: IFormatter и ISerializable (этот интерфейс будет подробнее рассмотрен ниже).

Вне зависимости от того, каким объектом Formatter мы будем пользоваться (пусть это будет даже специальный, созданный нами Formatter), главная его задача будет одной и той же — передавать всю информацию о состоянии объекта в процессе сериализации в место постоянного хранения, используя при этом какой-либо формат для ее записи. Эта информация обязательно должна включать в себя полное имя класса объекта (например, MyProject.MyClasses.Foo), имя сборки, содержащей в себе данный класс (дружественное имя, номер версии и, возможно, «сильное имя» сборки), а также необходимые сведения о состоянии объекта, для хранения которых используется объект SerializationInfo.

Объект Formatter участвует и в процессе десериализации: он использует информацию из постоянного места хранения (точнее, из потока) для воссоздания точной копии сериализованного объекта. Общая схема этого процесса представлена на рис. 11.21.

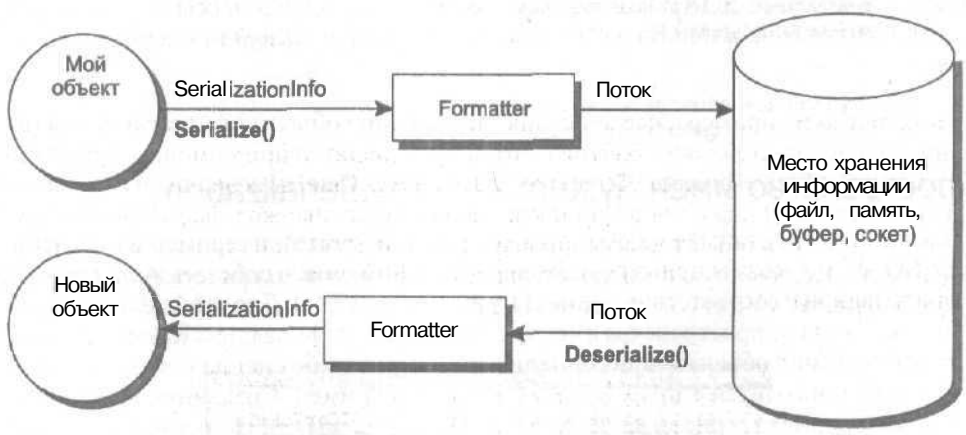


Рис 11.21. Процесс сериализации

Сериализация в двоичном формате

Сериализация объектов в двоичном формате производится при помощи типа BinaryFormatter, который определен в пространстве имен System.Runtime.Serialization.Formatters.Binary (а в физическом отношении — в библиотеке mscorlib.dll). Для записи объекта в место хранения и восстановления объекта по сохраненной информации используются два главных метода этого класса (табл. 11.16).

Таблица 11.16. Методы класса BinaryFormatter

Метод	Описание
Deserialize()	Десериализует поток байтов в объект
Serialize()	Сериализует объект (или граф из взаимосвязанных объектов) в поток

Кроме того, тип BinaryFormatter определяет набор свойств для уточнения некоторых особенностей процессов сериализации и десериализации. Однако в большинстве случаев эти свойства совершенно не нужны.

Давайте сериализуем наш шпионский автомобиль в двоичном формате (точнее, в файл CarData.dat). Выглядеть этот процесс может следующим образом:

```
using System.Runtime.Serialization.Formatters.Binary;

public static void Main()
{
        // Создаем объект JamesBondCar и выполняем с ним всякие действия
        JamesBondCar myAuto = new JamesBondCar("Fred", 50, false, true);
        myAuto.TurnOnRadio(true);
        myAuto.GoUnderWater();

        // Создаем поток для записи в файл
        FileStream myStream - File.Create("CarData.dat");

        // Помещаем объектный граф в поток в двоичном формате
        BinaryFormatter myBinaryFormat = new BinaryFormatter();
        myBinaryFormat.Serialize(myStream, myAuto);
        myStream.Close();
        ...
}
```

Как видно из примера, все действия по созданию объектного графа и перемещению последовательности байтов в поток производятся при помощи единственного метода — BinaryFormatter.Serialize(). В нашем случае мы сериализовали объект в файл на диске. Однако мы точно так же можем сериализовать файл в любой другой поток (то есть объект класса, производного от Stream), например в оперативную память. Полученную последовательность байтов можно увидеть, если открыть файл CarData.dat в Visual Studio (рис. 11.22).

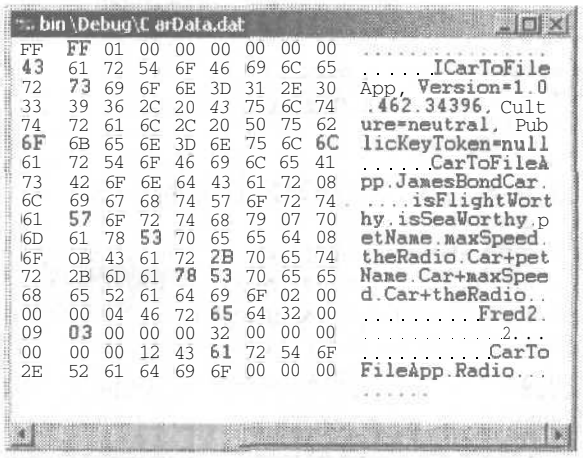


Рис. 11.22. Объект JamesBondCar сериализован в двоичном формате

Предположим, что нам потребовалось произвести обратную операцию — на основании сохраненной информации восстановить объект JamesBondCar. Для этого,

как мы уже говорили, необходимо использовать метод BinaryWriter.Deserializable(). Однако обратите внимание, что этот метод возвращает объект класса System.Object, и чтобы им можно было нормально пользоваться, мы должны явно привести этот объект к типу JamesBondCar:

```
// Считываем информацию об объекте из двоичного файла
myStream = FileOpenReade("CarData.dat");

JamesBondCar carFromDisk - (JamesBondCar)myBinaryFormat.Deserialize(myStream);

Console.WriteLine(carFromDisk.PetName + " is alive!");
carFromDisk.TurnOnRadio(true);
myStream.Close();
```

Обратите внимание, что при вызове метода Deserialize() мы должны передать ему объект класса, производного от Stream (в данном случае тот же самый поток из двоичного файла на диске). Таким образом, сериализация и десериализация в двоичном формате в .NET производится исключительно просто — только не забывайте помечать классы атрибутом [Serializable].

Сериализация в формате SOAP

Мы можем сериализовать объекты не только в двоичном формате, но и в формате SOAP. Для этого используется объект SoapFormatter. Как уже говорилось, прежде чем использовать этот объект, мы должны указать в нашем приложении ссылку на сборку Syste.Runtime.Serialization.Formatters.Soap.dll. Ниже приведен пример, в котором тот же самый объект JamesBondCar сериализуется в файл в XML-совместимом формате сообщений SOAP (если вам этот формат совершенно не знаком, пока не берите в голову — подробнее о SOAP мы будем говорить в главе 15). Код нового варианта нашего примера может выглядеть следующим образом:

```
using System.Runtime.Serialization.Formatters.Soap;

// Сохраняем тот же самый объект в формате SOAP
FileStream myStream - File.Create("CarData.xml");
SoapFormatter myXMLFormat - new SoapFormatter();
myXMLFormat.Serialize(myStream, myAuto);
myStream.Close();

// Восстанавливаем объект из файла SOAP
myStream - File.OpenRead("CarData.xml");
JamesBondCar carFromXML - (JamesBondCar)myXMLFormat.Deserialize(myStream);

Console.WriteLine(carFromXML.PetName + " is alive!");
myStream.Close();
```

Как можно убедиться, работа с SoapFormatter практически идентична работе с BinaryFormatter. Точно так же вызываются методы Serialize() и Deserialize() для сохранения объекта в месте хранения и его восстановления на основе сохраненной информации. Если мы откроем созданный нами файл CarData.xml, то сможем убедиться, что вся информация об объекте (включая информацию о его взаимосвязях с другими объектами) записана в тегах XML (рис. 11.23).

Код приложения CarToFile можно найти в подкаталоге Chapter 11.

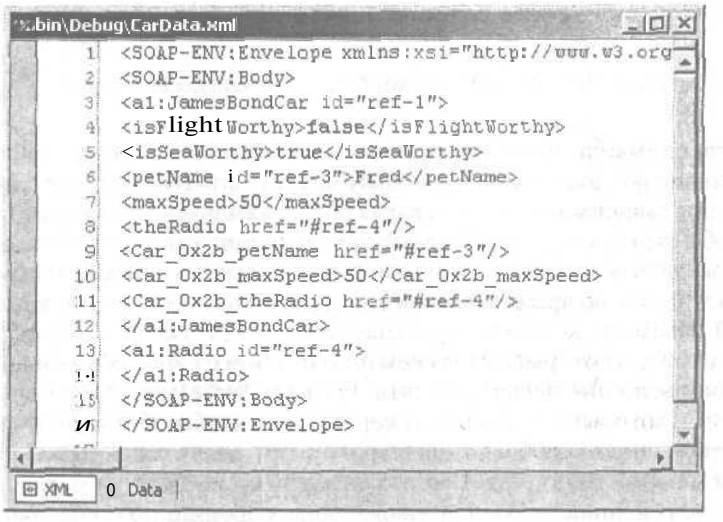


Рис. 11.23. Объект JamesBondCar сохранен в формате SOAP

Сериализация в пользовательском формате и интерфейс ISerializable

Как мы уже говорили, в принципе, не обязательно сериализовать объекты теми двумя способами, которые предусмотрены в виде объектов BinaryFormatter и SoapFormatter (хотя их возможностей в большинстве случаев вполне достаточно). Мы можем позаботиться о том, чтобы какой-то класс сериализовался специальным, предусмотренным только для него способом, определив в нем несколько специальных членов. Классы, которые определены в пространстве имен System.Runtime.Serialization, помогут нам в этом.

Первое, что мы должны сделать, — обеспечить реализацию в нашем классе интерфейса ISerializable. Определение этого интерфейса выглядит следующим образом:

```
// Для специальной сериализации какого-либо класса этот класс должен реализовать
// интерфейс ISerializable
public interface ISerializable
{
      public virtual void GetObjectData(SerializationInfo info, StreamingContext
                                                                  context);
}
```

Этот интерфейс определяет единственный метод GetObjectData(), который вызывается объектом Formatter в процессе сериализации. Реализация этого метода должна помещать в принимаемый в качестве параметра объект SerializationInfo все необходимые наборы имя — значение. Можно сказать, что SerializationInfo — это «мешок со свойствами» (property bag) — вещь хорошо знакомая программистам COM.

Объекты, для которых обеспечивается специальная сериализация, должны не только реализовать интерфейс ISerializable, но еще и обязательно определять конструктор со специальной сигнатурой:

```
// Конструктор со специальной сигнатурой необходим для процесса десериализации
class SomeClass
{
        private SomeClass (SerializationInfo si. StreamingContext ctx){...}
}
```

Обратите внимание, что область видимости этого конструктора определена как private. Это вполне допустимо, поскольку объект Formatter получает доступ к членам класса вне зависимости от того, какая область видимости для него установлена. Лучше, чтобы этот конструктор был определен именно как private — тогда ни у кого из случайных пользователей не возникнет идея создавать объекты таким несколько странным способом во время обычной работы (вне процесса десериализации).

Первый параметр, который принимает этот конструктор, — это объект класса SerializationInfo, в который мы можем поместить все пары имя — значение, представляющие состояние нашего объекта. В классе SerializationInfo определен метод AddValue(), который многократно перегружен, чтобы обеспечить возможность работы с переменными любого типа (строковыми, целочисленными, логическими и т. п.). Кроме того, в Seriallzatlonlnfo определено множество методов GetXXXX(), которые можно использовать для извлечения информации о состоянии объекта при его воссоздании. Мы еще познакомимся с этими методами в действии.

Второй параметр, который принимает этот конструктор, — объект StreamingContext, который используется для хранения информации о месте хранения информации об объекте. Наиболее важный член этого объекта — свойство State, для которого используются значения из перечисления StreamingContextStates (табл. 11.17).

Таблица 11.17. Значения перечисления StreamingContextStates

Значение	**Описание**
All	Указывает, что сериализованные данные могут быть переданы в любое место или получены из любого места
Clone	Указывает, что объект будет клонирован
CrossAppDomain	Указывает, что местом назначения или источником будет другой AppDomain
CrossMachine	Указывает, что место назначения или источник будут расположены на другом компьютере
CrossProcess	Указывает, что местом назначения или источником будет другой процесс на том же самом компьютере
File	Указывает, что местом назначения или источником будет файл
Other	Указывает, что место назначения или источник не определены
Persistence	Указывает, что место назначения или источник — постоянное хранилище. Им может, к примеру, служить база данных или файл. Обычно данные в таких хранилищах хранятся гораздо дольше времени жизни программного процесса, поэтому не следует помещать в постоянное хранилище объекты, для десериализации которых необходима ссылка на какие-то другие объекты текущего процесса (без информации об этих объектах)
Remoting	Указывает, что место назначения или источник неизвестны (например, удаленный компьютер или локальный компьютер)

Простой пример пользовательской сериализации

Еще раз повторим, что в подавляющем большинстве случаев стандартная сериализация, которая производится средствами встроенных в .NET объектов Formatter,

вполне приемлема. Однако для того, чтобы проиллюстрировать то, что было сказано выше, мы создадим новый вариант класса Car, который будет определен таким образом, чтобы к нему можно было применять пользовательскую сериализацию. Ничего особенно сложного при реализации GetObjectState() или специального варианта конструктора не потребуется. Достаточно, чтобы эти методы правильно взаимодействовали с внутренними переменными нашего объекта и сохраняли эту информацию в объекте SerializationInfo (или считывали ее из него). Выглядеть определение нашего класса может следующим образом:

```
public class CustomCarType : ISerializable
{
    public string petName;
    public int maxSpeed;
    public CustomCarType(string s, int i) { petName = s; maxSpeed = i; }

    // Передаем информацию о состоянии объекта объекту Formatter
    public void GetObjectData(SerializationInfo si, StreamingContext ctx)
    {
        // Каков тип нашего потока?
        Console.WriteLine("[GetObjectData] Context State: {0}",
                                        ctx.State.Format());

        si.AddValue("CapPetName", petName);
        si.AddValue("MaxSpeed", maxSpeed);
    }
    // А теперь позаботимся о специальной варианте конструктора
    private CustomCarType(SerializationInfo si, StreamingContext ctx)
    {
        // Какое тип нашего потока?
        Console.WriteLine("[ctor] Context State: {0}", ctx.State.Format());

        petName = si.GetString("CapPetName");
        maxSpeed = si.GetInt32("maxSpeed");
    }
}
```

Теперь мы можем произвести процессы сериализации и десериализации для нашего специального класса (результат показан на рис. 11.24):

```
public static int Main(string[] args)
{
    CustomCarType myAuto = new CustomCarType("Siddhartha", 50);
    Stream myStream = File.Create("CarData.dat");

    // Задействуем интерфейс ISerializable
    BinaryFormatter myBinaryFormat = new BinaryFormatter();
    myBinaryFormat.Serialize(myStream, myAuto);
    myStream.Close();

    myStream = File.OpenRead("CarData.dat");

    // Вызываем спецконструктор
    CustomCarType carFromDisk = (CustomCarType)myBinaryFormat.Deserialize(myStream);

    Console.WriteLine(carFromDisk.petName + " is alive!");
    return 0;
}
```

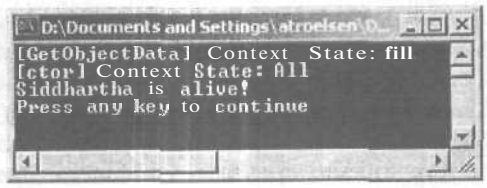


Рис. 11.24. Применение пользовательского варианта сериализации

Код приложения CustomSerialization можно найти в подкаталоге Chapter 11.

Приложение для регистрации автомобилей с графическим интерфейсом

В заключение нашего обсуждения вопросов, связанных с сериализацией объектов, мы рассмотрим в последней части этой главы простое, но вполне работоспособное приложение Windows, в котором применяется множество их тех технологий, с которыми мы познакомились. Это приложение позволит пользователю создавать список объектов Car (как массив ArrayList), которые будут отображаться на форме при помощи элемента управления DataGrid (рис. 11.25). Чтобы ничто не отвлекало нас от процессов сериализации, мы сделаем объект DataGrid доступным только для чтения.

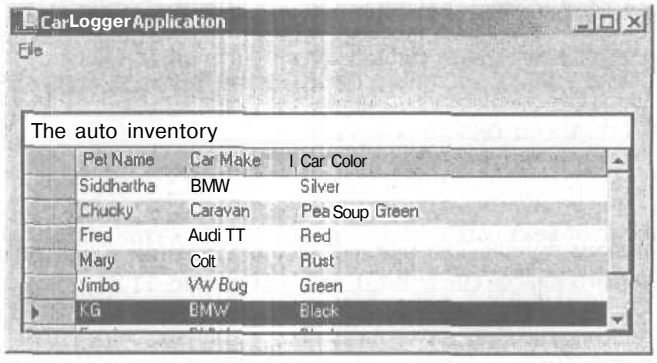


Рис. 11.25. Приложение для регистрации автомобилей

Все операции с массивом объектов Car будут производиться при помощи меню File. Варианты действий пользователя, которые будут доступны через это меню, представлены в табл. 11.18.

Вдаваться в особенности конструирования меню мы не будем, поскольку эти вопросы были подробно рассмотрены в главе 8. Первое, с чего мы начнем, — с определения самого класса Car. В книге уже встречалось немало разновидностей этого класса. Этот вариант будет отличаться предельной простотой:

```
[Serializable]
public class Car
{
    // Объявляем для простоты все переменные как public
```

```
        public string petName, make, color;

        public Car(string petName, string make, string color)
        {
                this.petName = petName;
                this.color = color;
                this.make = make;
        }
}
```

Таблица 11.18. Список элементов меню File в приложении CarLogApp

Элемент	Действие
Clear All Cars	Очищает массив ArrayList и обновляет DataGrid
Exit	Выход из приложения
Make New Car	Отображает специальное диалоговое окно для добавления пользователем нового объекта Car и обновляет DataGrid
Open Car File	Позволяет пользователю открывать существующий файл *.car и обновляет DataGrid. Файл *.car создается при помощи BinaryFormatter
Save Car File	Сохраняет все автомобили, отображаемые в DataGrid, в файле *.car

Далее нам потребуется добавить несколько объектов на нашу форму. Чтобы сэкономить время при размещении на форме элемента управления DataGrid, мы поместим его на форме при помощи Tool Box и установим его свойства при помощи графического интерфейса Visual Studio.NET. Самое важное свойство, которое нам потребуется установить, — свойство Readonly. Как мы договаривались, для него будет установлено значение true. Кроме того, нам потребуется задать для DataGrid его размеры и цветовую гамму (выбирайте на свой вкус).

Кроме того, мы должны обеспечить для нашей формы массив ArrayList, в котором будет производиться хранение объектов Car. Пусть конструктор формы изначально помещает в массив несколько объектов Car — чтобы при открытии формы в списке уже было несколько готовых к употреблению объектов. После добавления элементов в массив объект DataGrid нужно обновлять (чтобы в нем отобразились изменения). За это будет отвечать функция UpdateGrid(). А соответствующая часть определения формы будет выглядеть так:

```
public class mainForm : System.Windows.Forms.Form
{
        // Массив для хранения объектов Car
        private ArrayList arTheCars = null;

        public  mainForm()
        {
                InitializeComponent();
                CenterToScreen();

                // Добавляем несколько объектов Car в массив
                arTheCars = new ArrayList();
                arTheCars.Add(new Car("Siddhartha", "BMW", "Silver"));
                arTheCars.Add(new Car("Chucky", "Caravan", "Pea Soup Green"));
                arTheCars.Add(new Car("Fred", "Audi TT", "Red"));

                // Обновляем DataGrid
```

```
            UpdateGridO;
        }
```

Г

Метод UpdateGrid() прежде всего будет создавать объект класса System.Data.DataTable. Каждой строке в этой таблице будет соответствовать объект Car в массиве ArrayList. После того как объект DataTable создан и заполнен объектами автомобилей, он послужит источником данных для элемента управления DataGrid. Подробнее о классе DataTable и приемах работы с ним будет рассказано в главе 13, посвященной ADO.NET, а пока мы приведем код для UpdateGrid():

```
private void UpdateGridO
{
    if(arTheCars !- null)
    {
        // Создаем объект DataTable с именем Inventory
        DataTable inventory - new DataTable("Inventory");

        // Создаем объекты DataColumn
        DataColumn make - new DataColumn("Car Make");
        DataColumn petName - new DataColumn("Pet Name");
        DataColumn color = new DataColumnC'Car Color");

        // Добавляем объекты OataColumn в DataTable
        inventory.Columns.Add(petName);
        inventory.Columns.Add(make);
        inventory.Columns.Add(color);

        // Для каждого элемента массива создаем строку в таблице
        foreach(Car c in arTheCars)
        {
            DataRow newRow:
            newRow = inventory.NewRow();
            newRow["PetName"] - c.petName;
            newRow["Car Make"] - c.make;
            newRow["Car Color"] - c.color;
            inventory.Rows.Add(newRow);
        }

        // А теперь указываем объект DataTable в качестве источника данных
        // для элемента управления DataGrid
        carDataGrid.DataSource = inventory;
    }
}
```

Первое, что мы сделали, — создали новый объект DataTable с именем Inventory. В мире ADO.NET объект DataTable — это представление в оперативной памяти таблицы с данными. Очень часто объект DataTable создается как результат запроса на языке SQL, однако ничто не мешает нам, как в этом случае, создать его вручную.

После создания объекта DataTable нам необходимо определить столбцы, из которых будет состоять эта таблица. Для этой цели мы создали три объекта DataColumn (каждому из столбцов у нас соответствует переменная в классе Car) и добавили их в таблицу.

Таблица нам нужна не сама по себе, а с данными из массива. Поэтому следующее наше действие — создать для каждого объекта массива строку в таблице. По-

скольку класс ArrayList реализует интерфейс IEnumerable, проще всего сделать это при помощи конструкции foreach. И последнее, что нам осталось сделать, — указать созданный и заполненный данными объект DataTable в качестве источника данных для элемента управления DataGrid.

К этому моменту наше приложение уже работает! Если мы его запустим, то увидим, что в DataGrid на форме будет выводиться список автомобилей. Однако это, конечно, не все, что нам нужно от приложения, и поэтому мы продолжим работу.

Реализация добавления новых объектов Car

Добавление новых объектов Car пользователь будет производить при помощи отдельного модального диалогового окна AddCarDlg (рис. 11.26). Создание пользовательских диалоговых окон мы уже рассматривали в главе 10 и здесь повторяться не будем. Отметим только, что в CarAddDlg содержится объект TextBox (для ввода пользователем прозвища машины) и два объекта ListBox (для выбора цвета и модели машины соответственно).

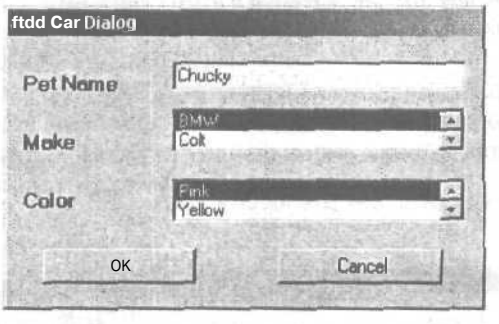


Рис. 11.26. Диалоговое окно Add Car

Для кнопки OK, конечно, мы определим свойство DialogResult как OK. Как уже говорилось в главе 10, это наделит кнопку целым рядом специальных функций. Кроме того, при нажатии на эту кнопку будет происходить создание нового типа Car со значениями, выбранными (или введенными) пользователем. Вот код для данного диалогового окна (служебный код, относящийся к настройке элементов графического интерфейса мы приводить не будем):

```
public class AddCarDlg : System.Windows.Forms.Form
{
        // Определяем класс для простоты доступа как public
        public Car theCar = null;

        ...

        protected void btnOK_Click (object sender, System.EventArgs e)
        {
                // Создаем новый объект Car при нажатии пользователем на кнопку OK
                theCar - new Car(txtName.Text, listMake.Text, listColor.Text);
        }
}
```

Диалоговое окно Add Car выводится при активации пользователем пункта меню Make New Car. Код для этого элемента меню будет выглядеть следующим образом:

```
protected void menuItemNewCar_Click (object sender. System.EventArgs e)
{
        // Открываем диалоговое окно и ожидаем нажатия кнопки OK
        AddCarDlg d - new AddCarDlg();
        if(d.ShowDialog() - - DialogResult.OK)
        {
                // Добавляем в массив только что созданный объект
                arTheCars.Add(d.theCar);
                UpdateGrid();
        }
}
```

Все просто и знакомо — при активации пункта меню открывается модальное диалоговое окно, а при нажатии в нем кнопки OK окно закрывается, и созданный объект Car помещается в массив ArrayList. Затем производится обновление DataGrid.

Код сериализации

Материал, с которым мы познакомились в этой главе, позволяет нам без труда создать код обработчиков событий Click для пунктов меню Save Car File и Open Car File. При выборе Save Car File программа должны создать новый файл и записать в него информацию сериализации объекта в двоичном формате (конечно, при помощи BinaryFormatter). Чтобы предоставить пользователю больше возможностей выбора, мы используем для выбора имени файла стандартное диалоговое окно, представленное типом System.Windows.Forms.SaveFi l eDialog. Внешний вид этого окна (хорошо всем знакомый) представлен на рис. 11.27.

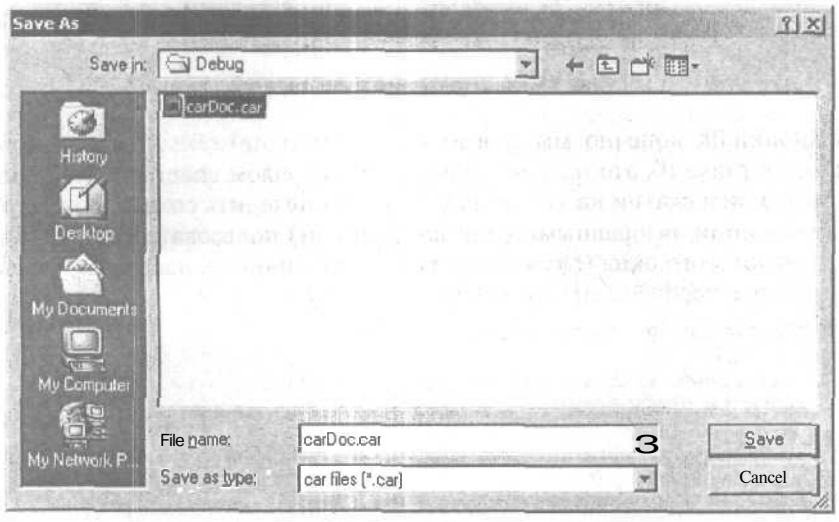


Рис. 11.27. Стандартное диалоговое окно для сохранения файла

Большую часть свойств SaveFi leDialog мы рассматривать не будем, отметим только один момент: мы воспользуемся свойством Filter для подсказки пользователю,

что файл с информацией о сериализации лучше сохранять с расширением *.car. Это свойство может принимать несколько значений, которые затем появятся в ниспадающих списках File name и Save as type. Полный код сериализации будет выглядеть следующим образом:

```
protected void menuItemSave_Click (object sender, System.EventArgs e)
{
        // Настраиваем свойства диалогового окна для сохранения файлов
        SaveFileDialog mySaveFileDialog = new SaveFileDialog();
        mySaveFileDialog.InitialDirectory - ".";
        mySaveFileDialog.Filter - "car files (*.car)|*.car|All files(*.*)|* *"·
        mySaveFileDialog.FilterIndex = 1;
        mySaveFileDialog.RestoreDirectory - true;
        mySaveFileDialog..FileName - "carDoc";

        // Сохраняем объекты автонобилей
        if(mySaveFileDialog.ShowDialog() - - DialogResult.OK)
        {
                Stream myStram - null;
                if((myStream - mySaveFileDiaog.OpenFile()) != null)
                {
                        BinaryFormatter myBinaryFormat = new BinaryFormatter();
                        myBinaryFormat.Serialize(myStream, arTheCars);
                        myStream.Close();
                }
        }
}
```

Метод SaveFileDialog.OpenFile() возвращает поток (объект Stream), который представляет файл, выбранный пользователем в диалоговом окне. Как мы уже могли убедиться в этой главе, B1naryFormatter ничего более и не требует.

Код десериализации (то есть обработчика события Click диалогового окна Open Car File) выглядит очень похоже. На этот раз мы используем еще одно заранее заготовленное диалоговое окно System.Windows.Forms.OpenFileDialog, настраиваем его соответствующим образом и получаем объект Stream для выбранного файла. После этого нам осталось воспользоваться методом B1naryFormatter.Deserialize() и поместить восстановленные объекты в массив:

```
protected void menuItemOpen_Click (object sender. System.EventArgs e)
{
        // Настраиваем свойства диалогового окна для открытия файлов
        OpenFileDialog myOpenFileDialog = new OpenFileDialog();
        myOpenFileDialog.InitialDirectory = ".";
        myOpenFileDialog.Filter = "car files (*.car)|*.car|All files(*.*)|*.*";
        myOpenFileDialog.FilterIndex - 1;
        myOpenFileDialog.RestoreDirectory = true;

        // Восстанавливаем объекты автомобилей
        if(myOpenFileDialog.ShowDialog() = = DialogResult.OK)
        {

                // Очищаем текущий массив
                arTheCars.Clear();

                Stream myStram = null;
                if((myStream = myOpenFileDiaog.OpenFile()) != null)
                {
```

```
                BinaryFormatter myBinaryFormat - new BinaryFormatter();
                arTheCars - (ArrayList)myBinaryFormat.Deserilize(myStream);
                myStream.Close();
                UpdateGridO:
            }
        ]
)
```

Замечательно! К этому моменту наше приложение уже умеет выполнять операции по сериализации (то есть сохранению объектов массива в файл) и десериализации (восстановлению их из файла в массив). Код для оставшихся пунктов меню очевиден:

```
protected void menuItemClear_Click (object sender, System.EventArgs e)
{
        arTheCars.Clear();
        UpdateGrid();
}

protected void menuItemExit_Click (object sender. System.EventArgs e)
{
        Application.Exit();
}
```

Код приложения CarLogApp можно найти в подкаталоге Chapter 11.

Подведение итогов

В самом начале этой главы определены типы Directory (DirectoryInfo) и File (FileInfo), которые предназначены для выполнения различных операций с физическими файлами и каталогами на диске. Далее в главе были рассмотрены классы, производные от Stream, — FileStream, MemoryStream и BufferedStream. Поскольку названия главных членов у этих классов совпадают, то эти классы в приложениях вполне взаимозаменяемы: нам не потребуется прилагать много усилий, чтобы, например, изменить место сохранения данных из файла в буфере оперативной памяти. Если нам потребуется обеспечить сохранение текстовых данных, наиболее простой и удобный способ сделать это — воспользоваться типами StreamWriter и StreamReader.

Последняя часть этой главы посвящена рассмотрению инфраструктуры .NET, предназначенной для сериализации и десериализации объектов. Мы познакомились с сохранением объектов в двоичном формате и формате SOAP. Несмотря на то что в большинстве случаев для сериализации объектов достаточно просто пометить соответствующий класс как [Serializable], в .NET предусмотрены средства для обеспечения пользовательских вариантов сериализации (интерфейс ISerializable и др.).

Взаимодействие с унаследованным программным кодом

12

Я думаю, что в процессе чтения этой книги вы неоднократно сравнивали возможности .NET и традиционных технологий создания приложений — COM, MFC, ATL и т. п. Скорее всего, вы убедились, что у .NET есть серьезные преимущества перед каждой из этих технологий. Однако мало кто из программистов может полностью забыть про COM, ATL, MFC, Visual Basic 6.0, Windows DNA и создавать приложения исключительно в среде .NET. Как правило, им приходится обеспечивать взаимодействие со многими тысячами строк унаследованного программного кода. И в .NET предусмотрены мощные и изящные средства реализации такого взаимодействия.

В начале этой главы мы покажем, как типы .NET могут взаимодействовать напрямую с Win32 API при помощи службы *PInvoke* (Platform Invoke). После этого настанет время обсудить взаимодействие .NET и COM, а также вопросы, связанные с RCW (Runtime Callable Wrapper). Далее в этой главе будет исследована обратная ситуация: как типы COM могут взаимодействовать с типами .NET при помощи CCW (COM Callable Wrapper). И в самом конце главы мы рассмотрим создание типов .NET, которые смогут взаимодействовать со службами, обеспечиваемыми средой выполнения COM+.

Главные вопросы совместимости

При создании сборок в .NET-совместимом компиляторе наш конечный продукт — это «управляемый код» (managed code), который предназначен для выполнения в среде CLR (Common Language Runtime). Управляемый код имеет ряд преимуществ по сравнению с традиционным двоичным кодом, таких как автоматическое управление памятью, единая система типов, самодокументирование сборок и т. п. Для сборок .NET предусмотрена специфическая и очень четкая внутренняя структура: помимо самих инструкций на языке IL (Intermediate Language, промежуточный язык) и метаданных типов, сборки содержат также манифест, в котором опи-

сываются все внутренние типы этой сборки и фиксируются все необходимые внешние сборки.

Очень часто сборки .NET должны успешно работать в мире сложных приложений, где значительную часть кода составляют классические COM-серверы. Код COM-серверов похож на код сборок .NET разве что расширениями имен файлов. Естественно, код модулей COM — двоичный, платформенно-зависимый (в отличие от полностью платформенно-независимого кода IL). COM-серверы работают с уникальным набором типов данных (BSTR, VARIANT и т. п.), содержание которых в разных языках программирования сильно различается. Помимо того что в каждом программном модуле COM должны быть реализованы достаточно сложные элементы (такие как фабрики классов, код IDL, записи в реестр), следует учитывать, что в мире COM необходимо отслеживать каждую ссылку на объект. Если мы ошибемся в ссылках, то это вполне может привести к таким неприятным последствиям, как утечка памяти.

В общем, между типами .NET и COM так мало общего, что трудно даже представить, как такие разные типы могут успешно взаимодействовать друг с другом. Однако, конечно же, в реальной работе возникает очень много ситуаций, когда необходимо обеспечить такое взаимодействие. Платформа .NET поддерживает следующие виды совместимости с унаследованным кодом:

- вызовы из типов .NET напрямую к модулям DLL, созданным на C (то есть обращения к Win32 API или пользовательским модулям DLL);
- вызовы из типов .NET к типам COM;
- вызовы из типов COM к типам .NET;
- вызовы из типов .NET к службам COM.

Как мы увидим далее в этой главе, в состав .NET SDK входит множество средств, которые позволяют «построить мост» между .NET и остальными архитектурами. Кроме того, в библиотеке базовых классов .NET также предусмотрено немало типов, которые предназначены исключительно для организации взаимодействия с унаследованным кодом.

Пространство имен System.Runtime.InteropServices

При использовании служб .NET для взаимодействия с унаследованным кодом мы будем явно или опосредованно работать с типами, определенными в пространстве имен System.Runtime.InteropServices. Наиболее важные типы этого пространства имен представлены в табл. 12.1.

Все перечисленные в табл. 12.1 типы — это атрибуты, которые используются для управления процессами передачи данных от типов .NET типам COM (и наоборот). Конечно, в пространстве имен System.Runtime.InteropServices определено также множество интерфейсов, перечислений и структур. Но мы будем рассматривать их не списком, а в момент их применения. Первое, с чего мы начнем, — с Platform Invocation Services (PInvoke).

Таблица 12.1. Некоторые типы пространства имен System.Runtime.InteropServices

Тип	Описание
ClassInterfaceAttribute	Используется для управления тем, как тип .NET будет представлять свои открытые члены клиентам COM
ComRegisterFunctionAttribute ComUnregistreFunctionAttribute	Могут быть связаны с пользовательскими методами. Определяют, что этот метод должен быть вызван при регистрации (или удалении регистрации) сборки для использования в среде COM
ComSourceInterfacesAttribute	Определяет список интерфейсов, которые являются источниками событий для класса
DispIdAttribute	Пользовательский атрибут, определяющий COM DISPID метода, открытой переменной или свойства
DllImportAttribute	Используется PInvoke (Platform Invocation Services — службами активизации платформ)
GuidAttribute	Используется для определения GUID для класса, интерфейса или библиотеки типов
IDispatchImplAttribute	Определяет, какую реализацию IDispatch должна использовать среда CLR при обнаружении двойных интерфейсов
InterfaceTypeAttribute	Определяет, как именно интерфейс .NET будет открыт для клиентов COM (в качестве производного от IDispatch или производного от IUnknown)
OutAttribute InAttribute	Используется для параметра или поля. Определяет, в каком направлении должны быть переданы данные — от вызываемого к вызывающему или от вызывающего к вызываемому
ProgIdAttribute	Пользовательский атрибут, который позволяет задавать ProgID для типа .NET

Взаимодействие с модулями DLL, созданными на C

Службы активизации платформ (Platform Invocation Services, PInvoke) обеспечивают возможность вызывать из кода .NET функции, реализованные в традиционных двоичных модулях DLL, написанных на C (не в COM-серверах). При помощи PInvoke программист .NET избавляется от массы проблем, связанных с реализацией вызова функций и экспорта возвращаемых ими данных вручную. PInvoke передает также параметры при вызове функций, транслируя типы данных .NET в их аналоги в традиционном двоичном коде.

Как мы уже говорили, одна из главных целей, которая преследовалась при создании библиотеки базовых классов .NET — спрятать от программиста низкоуровневые вызовы к Win32 API, Однако если нам все-таки потребовалось реализовать подобные вызовы — то PInvoke к нашим услугам. Кроме того, службы PInvoke могут быть использованы, конечно, и для доступа к функциям, определенным в пользовательских модулях DLL. Таким образом, если в нашем распоряжении остались библиотеки унаследованного кода, написанного на C, при помощи PInvoke мы можем без каких-либо проблем к ним обращаться.

Проиллюстрируем это примером. Пример будет выглядеть как класс C#, из которого будет производиться вызов к функции Win32 API MessageBox(). Вот он:

```
namespace PInvokeExample
{
using System;

        // Нужно для получения доступа к типам PInvoke
        using System.Runtime.InteropServices;

        public class PInvokeClient
        {
                // Функция Win32 MessageBox() живет в user32.dll
                [DllImport("user32")]
                public static extern int MessageBox(int hWnd, String pText, String
                                                    pCaption, int uType);

                public static int Main(string[] args)
                {
                        // Создаем несколько переменных .NET для передачи
                        // функции Win32 MessageBox()
                        String pText = "Hello World!";
                        String pCaption = "PInvoke Test";
                        MessageBox(0, pText, pCaption, 0);

                        return 0;
                }
        }
}
```

Процесс обращения к внешнему традиционному модулю DLL (то есть модулю DLL, написанному на C без всяких COM-элементов) начинается с объявления функции с ключевыми словами static и extern. (Это обязательно.) Обратите внимание, что при объявлении прототипа функции C мы должны, конечно, пользоваться типами данных .NET. Например, нельзя использовать типы данных char* и wchar_t* — вместо этого, как мы видим, используется тип System.String.

Кроме того, мы должны пометить этот прототип функции при помощи атрибута [Dllimport]. Как минимум, в значении этого атрибута мы должны указать имя модуля DLL, функцию из которого мы будем вызывать, как это у нас и сделано:

```
[DllImport("user32")]
public static extern int MessageBox(...);
```

Тип DllImportAttribute определяет значительное количество открытых переменных (полей), которые можно использовать для настройки процесса вызова внешней функции. Эти поля представлены в табл. 12.2.

Чтобы задать эти значения для текущего объекта DllImportAttribute, просто укажем каждую пару имя — значение в конструкторе класса. Конструктор DllImportAttribute принимает единственный параметр типа System.String:

```
class DllImportAttribute
{
        // Конструктор принимает переменную типа string, в которой будут содержаться
        // все пары имя поля - значение
        public DllImportAttribute (string val);
...
}
```

Понятно, что при таком подходе DllImportAttribute абсолютно безразлично, в каком наборе и в какой последовательности мы укажем эти пары имя — значение.

Строковое значение будет просмотрено целиком, и из него будут извлечены необходимые параметры.

Таблица 12.2. Поля типа DllImportAttribute

Поле	Описание
CallingConvention	Используется для определения соглашения о вызовах (calling convention), используемого при передаче аргументов метода
CharSet	Определяет, как именно будут передаваться вызываемой функции строковые данные
EntryPoint	Определяет имя или номер вызываемой функции
ExactSpelling	Если это поле будет иметь значение true, то никакие попытки служб PInvoke «догадаться», какое именно имя вызываемой функции имеется в виду (об этой особенности PInvoke — чуть позже), допускаться не будут: будет требоваться точное совпадение имени вызываемой функции с переданным именем
PreserveSig	Если это поле будет иметь значение true (это значение используется по умолчанию), сигнатура метода традиционной DLL *не будет* преобразована *в* сигнатуру метода .NET при возврате HRESULT и дополнительных аргументов [out, retval]
SetLastError	Если это поле имеет значение true, это значит, что вызывающий может вызывать метод Win32 GetLastError() для определения того, возникла ли ошибка при выполнении метода. По умолчанию для этого поля используется значение false

Поле ExactSpelling

Первое поле DllImport, которое заслуживает особого рассмотрения, — это поле ExactSpelling. Как нам уже известно (см. табл. 12.2), это поле определяет, будет ли требоваться точное соответствие между именем функции в .NET и именем функции в традиционном модуле DLL. Например, если сказать по правде, то функции с именем MessageBox() в Win32 API нет. Вместо нее есть две функции — версия ANSI (MessageBoxA()) и функция Unicode (MessageBoxW). Поэтому, анализируя наш пример, можно прийти к выводу, что раз он работает, ExactSpelling имеет значение в false, и это — значение этого поля по умолчанию (что совершенно верно). Если вы установите для этого поля значение true:

```
[DllImport("user32", ExactSpelling = true)]
public static extern int MessageBox(...); // Ой-ой...
```

то ничего хорошего не произойдет. Точнее, будет сгенерировано исключение EntryPointNotFoundException с объяснением, что точка входа MessageBox в модуле user32.dll не обнаружена.

Таким образом, при значении false поля ExactSpelling часть работы по розыску нужной функции службы PInvoke берут на себя. Например, в нашей ситуации суффикс A будет автоматически добавлен при работе в среде ANSI, а суффикс W — при работе в среде Unicode.

Поле CharSet

Чтобы явно указать кодировку для символьных данных при вызове внешней функции из кода традиционного модуля DLL, мы должны задать значение в поле CharSet

типа DllImportAttribute. Для этого поля используются значения из перечисления CharSet, представленные в табл. 12.3.

Таблица 12.3. Значения перечисления CharSet

Значение	Описание
Ansi	Определяет, что строковые значения должны быть переданы в однобайтовой кодировке ANSI
Auto	Определяет, что службы PInvoke сами должны определить, как передавать строковые значения (Unicode в WinNT/Win2000 и ANSI в Win 9x)
None	Это значение применяется по умолчанию. Оно значит, что не определена используемая кодировка и среда выполнения должна сделать это автоматически
Unicode	Определяет, что строковые значения должны передаваться в двухбайтовой кодировке Unicode

Чтобы принудительно передавать все строковые значения как Unicode (и подвергнуть свое приложение риску неправильной работы в Win95/98/ME), можно использовать такой код:

```
// Принудительно используем точное имя и кодировку Unicode
[DllImport("user32", ExactSpelling - true, CharSet=CharSet.Unicode)]
public static extern int MessageBoxW(...);
```

В большинстве случаев гораздо удобнее и безопаснее задать для поля CharSet значение CharSet.Auto или просто оставить значение по умолчанию (None). В этой ситуации код будет корректно выполняться на разных платформах и, таким образом, станет более переносимым.

Поля CallingConvention и EntryPoint

Последние два поля DllImportAttribute, которые мы рассмотрим, — это поля CallingConvention и EntryPoint. Как вы, наверное, знаете, для функций Win32 API можно использовать множество специальных параметров (typedef), которые определяют то, каким образом параметры будут переданы функции: объявление C (C declaration), быстрый вызов, стандартный вызов и т. п. Эти значения для вызова через PInvoke можно определить с помощью поля CallingConvention, для которого используются значения из одноименного перечисления. В этом перечислении предусмотрены такие значения, как Cdecl, Winapi, StdCall и т. п. По умолчанию используется значение StdCall, и, как правило, в большинстве ситуаций этого вполне достаточно (поскольку это соответствует наиболее распространенному соглашению о вызовах, используемому в Win32).

Последнее поле (по порядку рассмотрения, но не по важности) — это поле EntryPoint. По умолчанию значение этого поля совпадает с именем функции-прототипа в C#. Таким образом, в нашем примере значение этого поля будет автоматически установлено в MessageBoxW:

```
// Значение поля EntryPoint будет автоматически установлено в 'MessageBoxW'
[DllImport("user32", ExactSpelling - true, CharSet=CharSet.Unicode)]
public static extern int MessageBoxW(...);
```

Однако не всегда в коде .NET можно использовать то же имя, которое нужно использовать для вызова функции из внешней традиционной библиотеки DLL

(самая простая ситуация — функция с тем же названием уже определена в коде .NET). В этой ситуации мы можем явно указать имя вызываемой внешней функции в качестве значения поля EntryPoint:

```
public class PInvokeClient
{
        // Функция .NET DisplayMessage() будет вызывать функцию MessageBox()
        // из традиционной DLL
        [DllImport("user32", ExactSpelling - true, CharSet=CharSet.Unicode, EntryPoint =
                                                                    "MessageBoxW")]
        public static extern int DisplayMessage(int hWnd, String pText, String pCaption,
                                                                    int uType);

        public static int Main(string[] args)
        {
                String pText - "Hello World!";
                String pCaption = "PInvoke Test";

                // Реально будет вызвана функция MessageBox()
                DisplayMessage(0, pText, pCaption, 0);
                return 0;
        }
}
```

Код приложения PinvokeExample можно найти в подкаталоге Chapter 12.

Взаимодействие .NET и COM

Следующий тип взаимодействия с унаследованным кодом, который мы должны рассмотреть, — это взаимодействие с модулями COM. Очень часто при создании приложений .NET оказывается, что им приходится взаимодействовать с существующими COM-серверами. Естественно, для решения этой задачи необходимо обеспечить какой-то промежуточный уровень, который транслировал бы вызовы .NET в вызовы COM (представляя типы COM в виде типов .NET). В идеальной ситуации процесс подобного преобразования должен быть совершенно прозрачен для типов .NET, чтобы они могли обращаться к типам COM точно так же, как и к другим типам .NET.

Роль подобного промежуточного уровня в .NET выполняют службы RCW (Runtime Callable Wrapper, вызываемая оболочка времени выполнения). Каждому классу COM (соклассу), к которому производятся вызовы из клиента .NET, необходима своя оболочка — свой RCW. Таким образом, если мы работаем с одним приложением .NET, которое обращается к трем соклассам COM, нам потребуются три отдельных RCW, которые будут преобразовывать вызовы .NET в вызовы COM. Общая схема этого процесса представлена на рис. 12.1.

Еще раз отметим, что на каждый COM-сервер нужен один модуль RCW, вне зависимости от того, к какому количеству интерфейсов в этом COM-сервере обращается клиент .NET. Каждый модуль RCW содержит в себе информацию об одном-единственном модуле COM и только о нем. При этом модуль RCW используется также для отслеживания ссылок на объекты COM, созданные из .NET, обеспечивая при этом сборку мусора и прочие принципиальные возможности .NET.

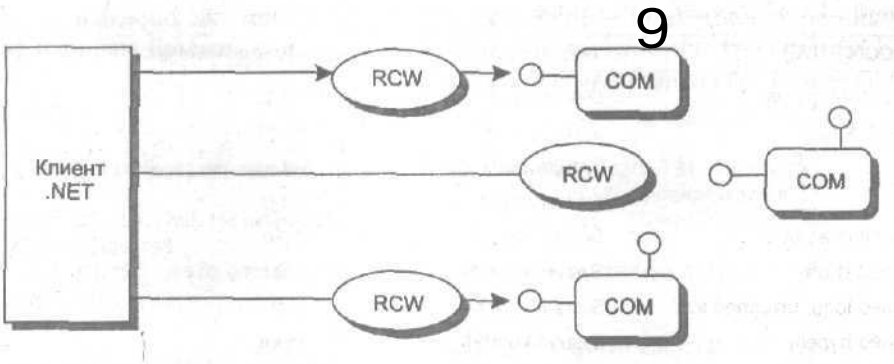


Рис. 12.1. RCW функционирует как промежуточный уровень между клиентом .NET и COM-сервером

Я думаю, что вас обрадует сообщение, что RCW создавать вручную не нужно — это производится автоматически при помощи утилиты tlbimp.exe (от type library importer — импортер библиотеки типов). Еще одна новость (впрочем, ожидаемая), которая может поднять вам настроение, заключается в том, что для нормального взаимодействия с клиентами .NET в унаследованные COM-классы не придется вносить никаких изменений. Всю необходимую работу берут на себя модули RCW. О том, как они это делают и каким образом можно их настроить — в следующих разделах.

Представление типов COM как типов .NET

За то, чтобы типы COM представлялись типам .NET как такие же модули .NET, отвечают программные модули RCW. Предположим, что в модуле COM определен метод (входящий в интерфейс), который описан в IDL:

```
// Определение метода COM в IOL
HRESULT DisplayThisString([in] BSTR msg);
```

Модуль RCW представляет этот метод клиенту .NET следующим образом:

```
// Представление метода COM в C#
void DisplayThisString(String msg);
```

У подавляющего большинства типов COM (включая [oleautomation]-совместимые типы) есть соответствующие им типы в .NET. Эти соответствия приведены в табл. 12.4.

Естественно, при работе с указателем в IDL (например, int* вместо int) этот указатель будет отображаться в соответствующий базовый класс (System. Int32/int). Ниже в этой главе, когда мы создадим наш собственный сервер COM ATL, мы рассмотрим вопросы, связанные с представлением более интересных типов данных, таких как SAFEARRAY и перечисления COM.

Управление ссылками на объекты сокласса

Как мы уже говорили, еще одна важная «зона ответственности» RCW заключается в отслеживании ссылок на сокласс COM. При обычном использовании соклассов в среде COM в этот процесс вовлечены и клиенты сокласса, и сам сокласс, а управление ссылками производится при помощи вызовов методов AddRef() и Release(). Классы COM саморазрушаются, когда более нет внешних ссылок на них.

Таблица 12.4. Соответствие встроенных типов COM с типами .NET

Тип COM (IDL)	Тип .NET	Псевдоним в C#
char, boolean, small	System.SByte	sbyte
wchar_t, short	System.Int16	short
long, int	System. Int32	int
hyper	System. Int64	long
unsigned char, byte	System.Byte	byte
unsigned short	System. UInt16	ushort
unsigned long, unsigned int	System. UInt32	uint
unsigned hyper	System.UInt64	ulong
single	System.Single	float
double	System.Double	double
VARIANT_BOOL	Нет	bool
HRESULT	System.Int32	int
BSTR	System.String	string
LPSTR или char*	System.String	string
LPWSTR или wchar_t*	System.String	string
VARIANT	System.Object	object
DECIMAL	System. Decimal	Нет
DATE	System. DateTime	Нет
GUID	System.Guid	Нет
CURRENCY	System. Decimal	Нет
IUnknown*	System.Object	object
IDispatch*	System.Object	object

Однако в мире .NET такая схема не используется, поэтому для обеспечения нормального взаимодействия с COM-сервером клиент .NET должен был бы в нужный момент времени производить вызов метода `Release()`. К счастью, этого не требуется: как обычно, всю черновую работу берет на себя модуль RCW. Он производит кэширование всех ссылок на интерфейсы и в нужный момент производит вызов метода `Release()` для сервера COM, на который больше нет активных ссылок со стороны клиентов .NET. В результате клиентам .NET нет необходимости явно производить вызовы методов `AddRef()`, `Release()` или `QueryInterface()`.

Сокрытие низкоуровневых интерфейсов COM

Поскольку модуль RCW должен представить типы COM для клиента .NET точно так же, как будто это обычные типы .NET, он должен уметь также скрывать низкоуровневые интерфейсы COM. Во многих отношениях поведение модуля RCW в этой ситуации напоминает Visual Basic 6.0.

Например, когда мы создаем класс COM, который поддерживает интерфейс `IConnectionPointContainer` (и поддерживает подчиненный объект или два вспомогательных интерфейса `IConnectionPoint`), сокласс имеет возможность пересылать сообщения о событиях обратно клиенту COM. При использовании клиента COM,

созданного на C++, нам потребуется создать приемник события, который реализует интерфейс [source], получить ссылку на интерфейс IConnectionPoint, вызвать метод Advise() и проделать еще множество шагов по установлению соединения.

При создании клиента COM на Visual Basic 6.0 большая часть этого процесса полностью сокрыта (достаточно использовать ключевое слово WithEvents). Модуль RCW точно так же прячет всю рутину COM при создании клиента .NET. Все низкоуровневые интерфейсы COM-сервера полностью сокрыты от клиента COM, поэтому он видит (и, соответственно, может с ними работать) только пользовательские интерфейсы, реализованные в соклассе. Некоторые из интерфейсов COM, которые скрывает модуль RCW, представлены в табл. 12.5.

Таблица 12.5. Скрытые интерфейсы COM

Интерфейс	Описание
IClassFactory	Обеспечивает независимый от языка и местонахождения метод активации класса COM
IConnectionPointContainer IConnectionPoint	Обеспечивает возможность отправки соклассом событий обратно клиенту
IDispatch IDispatchEx IprovideClassInfo	Используется для реализации позднего связывания
IEnumVariant	Обеспечивает возможность представления соклассом собственного набора внутренних типов
IErrorInfo ISupportErrorInfo ICreateErrorInfo	Обеспечивают возможность для клиентов и соклассов COM генерации сообщений об ошибках и реагирования на такие сообщения
IUnknown	Управляет счетчиком ссылок и позволяет клиенту получать ссылку на конкретный интерфейс сокласса

Некоторое теоретическое представление о роли RCW мы уже получили. Настало время познакомиться с особенностями его применения на практике с помощью Visual Basic, 6.0 (в нем мы создадим простой COM-сервер) и C# (для создания клиента .NET). Ниже в этой главе мы создадим для наших целей более сложный COM-сервер с использованием ATL 3.0.

Создание простого COM-сервера в Visual Basic 6.0

Наша задача сейчас — создать очень простой COM-сервер, который будет называться Painfully Simple VB COM Server. Первое, что нам нужно сделать — открыть Visual Basic 6.0 и создать новый проект ActiveX DLL (рис. 12.2). Если вы пришли из мира C++ и ATL и вам не очень понятно, что же такое мы делаем, то поясним, что мы воспользовались шаблоном для создания встроенного в процесс (in-process) COM-сервера.

Далее воспользуемся окном свойств и переименуем проект в PainfullySimpleVBCOMServer (или что-нибудь более короткое), а исходный класс - в CoCalc. Эта информация будет использована для создания ProgID нашего COM-сервера при помощи стандартной схемы Имя_сервера.Имя_объекта.

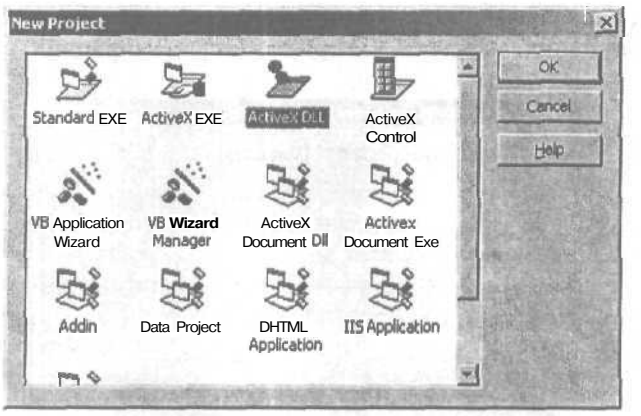


Рис. 12.2. Создаем COM-сервер в Visual Basic

Далее откроем окно кода для класса VB 6.0 CoCalc и добавим следующее вполне обычное определение функции:

```
' Это не что иное, как метод
' интерфейса по умолчанию _CoCalc!
'
Public Function Add(ByVal x As Integer. ByVal y As Integer) As Integer
        Add - x + y
End Function
```

После этого сохраните проект и откомпилируйте созданный COM-сервер с помощью меню File (Файл) ▸ Make (Создать). При этом созданный нами COM-сервер будет автоматически зарегистрирован в реестре нашей операционной системы. ProgID для созданного нами COM-сервера показан на рис. 12.3.

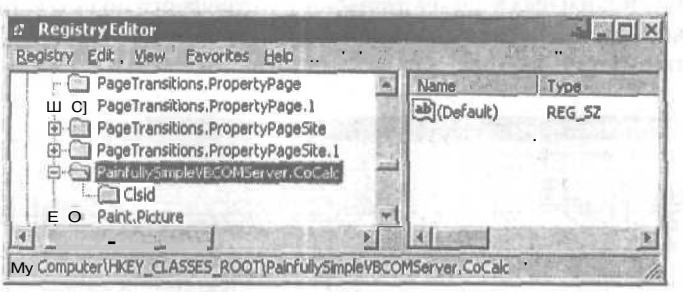


Рис. 12.3. ProgID нашего COM-сервера

Таким образом, результатом нашей работы стал COM-сервер с единственным классом (CoCalc), который реализует единственный интерфейс (он же интерфейс по умолчанию), который называется _CoCalc. В мире Visual Basic нам всегда совершенно бесплатно предоставляется такой интерфейс.

Перед тем как закрыть наш суперпроект, сделаем еще одно дело: выберем в меню Project (Проект) пункт Properties (Свойства) и установим переключатель Version Compatibility (Совместимость версий) в положение Binary Compatibility (Совместимость

по двоичному коду) — см. рис. 12.4. Это нужно сделать для того, чтобы Visual Basic перестал компилировать новые GUID при каждой компиляции.

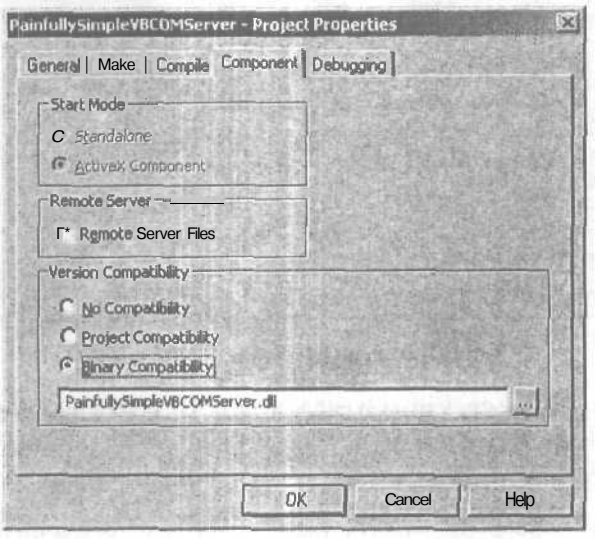


Рис. 12.4. Прекращаем генерацию GUID в Visual Basic

Код приложения PainfullySimpleVBCOMServer можно найти в подкаталоге Chapter 12.

Что находится в IDL нашего COM-сервера

Откроем OLE/COM Object Viewer и найдем там ProgID нашего COM-сервера (рис. 12.5). Мы увидим имя пользовательского интерфейса по умолчанию (CoCalc), а также множество служебных интерфейсов COM, которые были созданы Visual Basic автоматически.

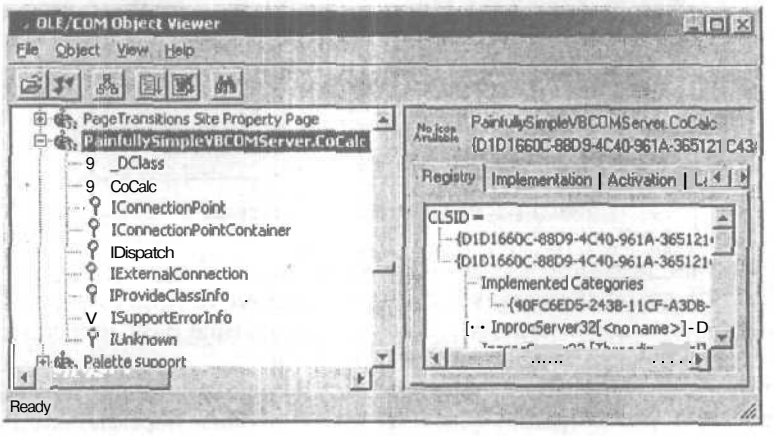


Рис. 12.5. Наш COM-сервер в OLE/COM Object Viewer

Чтобы просмотреть код IDL, щелкнем правой кнопкой мыши для нашего COM-сервера и в контекстом меню выберем View Type Information (Просмотр информации типов). Среди прочего кода IDL можно найти следующие строки, относящиеся к интерфейсу по умолчанию ([default]):

```
[ odl, uuid(DDA5B80E-8DA4-45DF-B8FF-B6BFFFBCD9E6),
version(1.0), hidden, dual, nonextensible, oleautomation ]
Interface _CoCalc : IDispatch
{
        [id(0x60030000)]
        HRESULT Add([in] short x, [in] short y, [out, retval] short * );
};

[uuid(D1D1660C-88D9-4C40-961A-365121C43AF1), version(1.0)]
coclass Cocalc
{
        [default] interface _CoCalc;
};
```

Как мы могли убедиться, Visual Basic всегда определяет пользовательский интерфейс как [dual]. В результате к нашему соклассу можно будет обращаться для работы со скриптами через интерфейс IDispatch с помощью самых разных языков.

Простой COM-сервер создан. Настало время заняться клиентом для этого COM-сервера. Вначале мы создадим его в том же самом Visual Basic 6.0.

Создаем простой клиент COM в Visual Basic 6.0

Откроем вновь Visual Basic 6.0 и выберем на этот раз шаблон Standard EXE. После этого при помощи меню Project (Проект) ▸ References (Ссылки) добавим в наш шаблон ссылку на Painfully SimpleVBCOMServer (рис. 12.6).

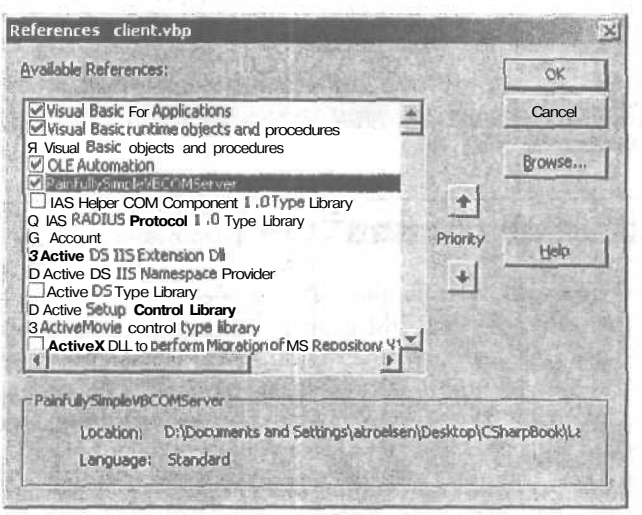


Рис. 12.6. Добавляем ссылку на наш COM-сервер

Клиент у нас будет похож на сервер своей предельной простотой. В нем будут предусмотрены два текстовых поля (объекта TextBox), единственная кнопка (объект Button) и две текстовые надписи рядом с текстовыми полями. Выглядеть все это может примерно так, как показано на рис. 12.7.

Единственный код, который нам вообще понадобится добавить на форму — это код для события кнопки Click. Мы создадим экземпляр сокласса и перешлем ему значения, которые будут браться, естественно, из текстовых полей. Для простоты мы будем отображать возвращаемый результат при помощи MessageBox:

```
Private Sub btnAdd_Click()
      Dim c As New CoCalc
      msgBox c.Add(txtNumb1, txtNumb2)
End Sub
```

Можно сказать, что мы создали гармонично дополняющие друг друга COM-сервер и COM-клиент. Следующая наша задача — создать клиент C#, который будет обращаться к тому же COM-серверу.

Код приложения PainfullySimpleVBCOMClient можно найти в подкаталоге Chapter 12.

Импорт библиотеки типов

Первый шаг, который мы должны сделать, прежде чем обращаться к COM-серверу из .NET, — создать промежуточный класс, который будет содержать в себе всю необходимую информацию для передачи запроса COM-серверу (об этом мы говорили, когда обсуждали RCW). Создание такого промежуточного класса производится при помощи утилиты tlbimp.exe (type library importer — импортер библиотеки типов). Вначале перейдем в командной строке в тот каталог, в котором находится двоичный модуль COM-сервера, а затем выполним команду следующего вида;

```
tlbimp PainfullySimpleVBCOMServer.dll /out:SimpleAssembly.dll
```

Давайте откроем только что созданную нами сборку при помощи ILDasm.exe (рис. 12,8). Обратите внимание, что всем элементам COM-сервера автоматически подобраны эквиваленты .NET.

Добавление ссылки на сборку

Главная задача созданной нами сборки .NET — обеспечить передачу запросов из модулей .NET традиционному COM-серверу. Для того чтобы в этом удостовериться, мы создадим COM-клиент на C# (он будет называться CSharpCalcClient). Первое, что нам нужно будет сделать — создать новое консольное приложение C# и присвоить ему имя. Затем, конечно же, нам потребуется добавить ссылку на созданную нами при помощи tlbimp.exe сборку. Добавление ссылки производится через диалоговое окно Add Reference (Добавить ссылку). После добавления ссылки сборка SimpleAssembly должна появиться в окне Solution Explorer (рис. 12.9).

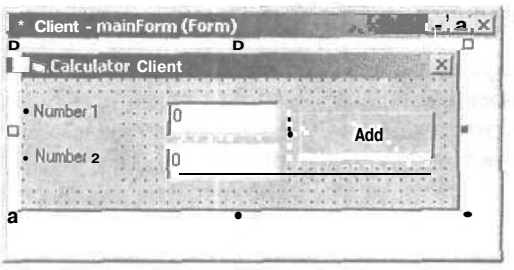


Рис. 12.7. Интерфейс нашего COM-клиента

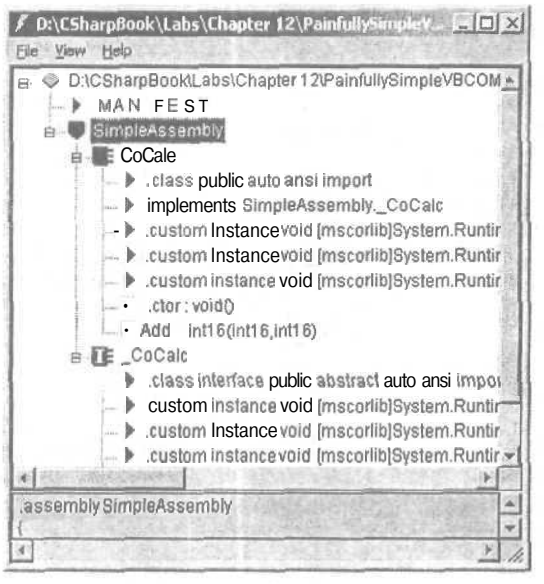


Рис. 12.8. Типы созданной нами сборки

Рис. 12.9. При создании клиента .NET должна быть добавлена ссылка на промежуточную сборку

Раннее связывание с COM-классом CoCalc

Мы уже добавили прямую ссылку на созданную нами сборку SimpleAssembly.dll и поэтому можем воспользоваться простым ранним связыванием. В приведенном ниже коде обратите внимание, что для клиента C# CoCalc — это обычный тип .NET, для обращения к которому ничего специального не требуется. В действительности же, конечно, запросы с CoCalc будут передаваться SimpleAssembly COM-серверу.

```
namespace CSharpCalcClient
{
        using System;

        // Добавим для упрощения доступа к CoCalc
        using SimpleAssembly;

        public class CalcClient
        {
                public static int Main(string[] args)
                {
                        // Создаем объект CoCalc
                        CoCalc c = new CoCalcO:

                        // Производим операции с числами через COM-сервер
                        Console.WriteLine("30 + 99 is: " + c.Add(30, 99));

                        return 0;
                }
        }
}
```

Как мы видим, все члены интерфейса по умолчанию ([default]) сокласса представляются напрямую как члены класса CoCalc. Если нам потребуется явно обратиться к какому-либо интерфейсу, это можно сделать при помощи следующего кода (мы будем обращаться к тому же _CoCalc):

```
public class CalcClient
{
        public static int Main(string[] args)
        {
                // Создаем объект CoCalc
                CoCalc c = new CoCalc();

                // Явным образом получаем ссылку на интерфейс
                _CoCalc icalc - c:
                Console.WriteLine("icalc.Add(9, 80));

                return 0:
        }
}
```

Код приложения CSharpCalcClient можно найти в подкаталоге Chapter 12.

Раннее связывание при помощи Visual Studio.NET

Надо сказать, что все можно сделать и проще — как обычно, средствами Visual Studio.NET. Среда разработки позволяет нам просто добавить COM-сервер при помощидиалоговогоокнаAddReference(Добавитьссылку)(рис. 12.10).

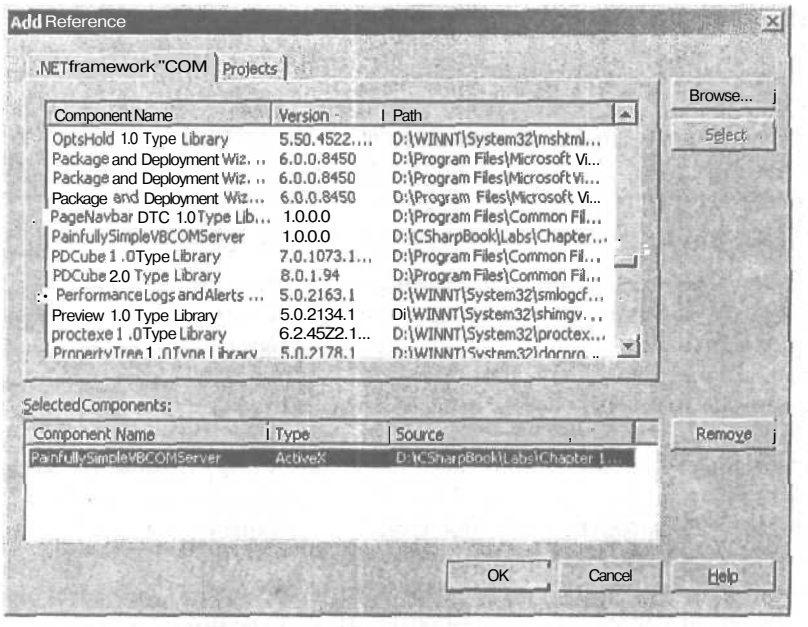


Рис. 12.10. Обращение к COM-серверу средствами Visual Studio.NET

В этом случае все будет сделано автоматически: будет вызвана утилита tlbimp.exe, которая создаст новую сборку в каталоге Debug (или Release), и ссылка на эту сборку будет добавлена в проект. В оставшейся части этой главы мы будем пользоваться для скорости именно графическими средствами Visual Studio.NET.

Позднее связывание с соклассом CoCalc

Мы уже обсуждали в главе 7, что в пространстве имен System.Reflection предусмотрены способы получения информации о типах сборки непосредственно во время выполнения. В мире COM аналогичные возможности реализуются при помощи набора стандартных интерфейсов (ITypeLib, ITypeInfo и т. д.). Связывание клиента с COM-сервером во время выполнения программы (в противоположность связыванию во время компиляции) называется поздним связыванием.

Прежде всего нужно сказать, что когда у нас есть выбор между ранним и поздним связыванием, всегда следует выбирать раннее. Однако в некоторых ситуациях без позднего связывания не обойтись. Например, некоторые унаследованные COM-серверы устроены таким образом, что они вообще не предоставляют никакой информации о типах. В этом случае утилита tlbimp.exe и модули RCW будут совершенно бесполезны. Однако способы организации взаимодействия .NET-клиентов и COM-серверов существуют и для такого случая. Мы должны организовать такое взаимодействие при помощи позднего связывания и типов из пространства имен System.Reflection.

Процесс позднего связывания начинается с получения клиентом от сокласса ссылки на интерфейс IDispatch. Этот стандартный интерфейс COM определяет четыре метода, два из которых нам интересны в настоящий момент. Первый ме-

тод — GetIDsOfNames(). Он позволяет клиенту, применяющему метод позднего связывания, получать числовое значение (DISPID), используемое для идентификации метода, который клиент собирается вызвать.

В COM IDL идентификатор DISPID для члена назначается при помощи атрибута [id]. Если мы обратимся при помощи OLE/COM Object Viewer к коду IDL, который был сгенерирован для нашего COM-сервера средой Visual Basic, мы сможем увидеть DISPID для метода Add():

```
[id(0x60030000)] HRESULT Add( [in] short x, [in] short y, [out. retval] short* ):
```

Числовое значение, помеченное атрибутом [id], — это именно то, что метод GetIDsOfNamesO возвратит клиенту, применяющему метод позднего связывания. Получив это значение, клиент сможет воспользоваться вторым интересующим нас методом — Invoke(). Этот метод интерфейса IDispatch принимает несколько параметров, один из которых — DISPID вызываемого метода.

Кроме того, метод Invoke() принимает массив типов COM VARIANT, представляющих параметры, передаваемые методу. В ситуации с методом AddC) этот массив будет содержать два значения типа short. Последний параметр Invoke() — это еще один тип VARIANT, который будет представлять возвращаемое клиенту значение (в нашем случае опять-таки short).

Клиент .NET, использующий позднее связывание, применяет вышеописанную схему, конечно, не напрямую (напрямую без RCW он не может обратиться к COM-серверу), а при помощи типов из пространства имен System.Reflection. Давайте создадим еще один клиент C#, который будет использовать позднее связывание для вызова метода AddO. Обратите внимание, что этому приложению уже не будут нужны никакие промежуточные сборки, создаваемые при помощи утилиты tlbimp.exe:

```
using System;
using System.Reflection;

public class LateBinder
{
        public static int Main(string[] args)
        {
                // Вначале получаем ссылку на интерфейс IDispatch от сокласса
                Type calcObj =
                        Type.GetTypeFromProgID("PainfullySimpleVBCOMServer.CoCalc");

                object calcDisp = Activator.CreateInstance(calcObj);

                // Создаем массив параметров
                object[] addArgs = { 100, 34 };

                // Вызываем метод Add() и получаем результат
                object sum = null;
                sum = calcObj.InvokeMember("Add", BindingFlags.InvokeMethod, null,
                                                        calcDisp, addArgs);

                // Выводим результат
                Console.WriteLine("Late bound adding:\n100 + 24 is: {0}", sum);
                return 0;
        }
}
```

Код приложения CSharpLateBoundClient можно найти в подкаталоге Chapter 12.

Особенности созданной нами сборки

Теперь, когда мы познакомились с тем, как можно обращаться к COM-серверу из кода .NET, необходимо уточнить некоторые важные детали. Давайте загрузим созданную нами при помощи tlbimp.exe сборку SimpleAssembly.dll в ILDasm.exe и рассмотрим повнимательнее ее содержимое (точнее, содержимое ее манифеста). Как и в любой сборке, мы сможем обнаружить в ней ссылку на сборку mscorlib.dll (библиотеку базовых классов .NET), за которой следует информация о версии.

Действительно неожиданные вещи обнаруживаются при обращении к атрибутам .NET. Среди них можно встретить ссылки на типы GuldAttribute и ImportedFromTypeLibAttribute. Если мы посмотрим на значение ImportedFromTypeLibAttribute, то сможем заметить жестко прописанный в файловой системе путь к COM-серверу (рис. 12.11).

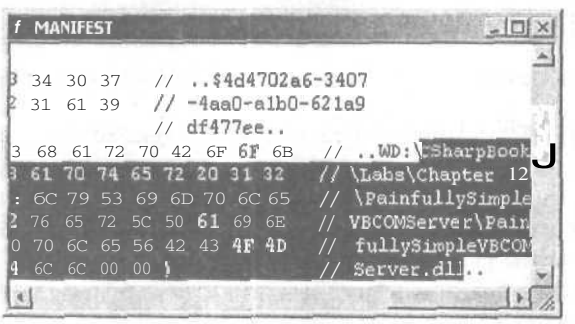


Рис. 12.11. Путь в файловой системе к двоичному файлу COM-сервера жестко определен в сборке как значение атрибута ImportedFromTypeLibAttribute

Из этого следует грустный вывод: если COM-сервер будет перемещен в другое место или переименован, наше приложение перестанет работать. Для восстановления работоспособности нам придется заново создавать SImpleAssembly.dll.

И еще один момент, на который можно обратить внимание. На рис. 12.12 представлено значение атрибута GuldAttribute.

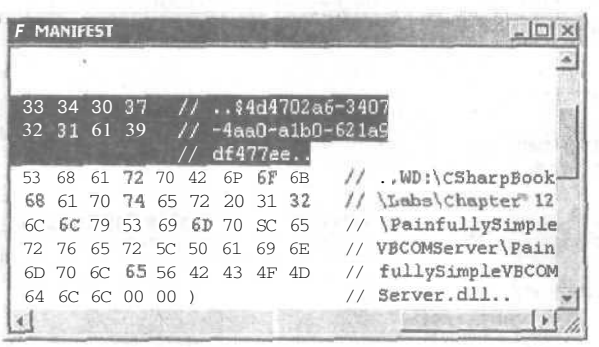


Рис. 12.12. Значение атрибута GuidAttribute — это GUID COM-сервера

Это значение — именно то, которое Visual Basic использовал в качестве GUID для библиотеки типов (LIBID). Если мы откроем наш COM-сервер в OLE/COM Object Viewer, то вы сможете убедиться, что это именно так (рис. 12.13).

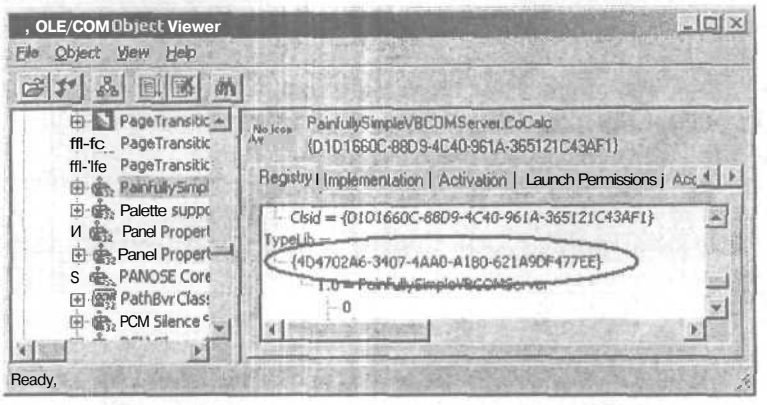


Рис. 12.13. Проверка GUID COM-сервера при помощи OLE/COM Object Viewer

Теперь давайте рассмотрим, что у нас имеется для самого класса CoCalc. Помимо информации о методе Add(), в манифесте содержится также информация о конструкторе этого класса по умолчанию. Это вполне понятно: поскольку RCW представляет COM-сервер в виде обычного класса .NET, то у этого класса должен быть конструктор. Если мы рассмотрим класс CoCalc в ILDasm.exe, то *сможем* увидеть также информацию о базовом классе для CoCalc (это будет класс System.Object) и реализованных интерфейсах (_CoCal c):

```
.class public auto ansi import CoCalc
extends [mscorlib] System.Object
implements SimpleAssembly._CoCalc
{
...

} // Конец информации о CoCalc
```

Чтобы разобраться, а как же все-таки выглядит представление COM-сервера в виде класса .NET, рассмотрим вариант метода Main(), в котором производится получение информации о COM-сервере при помощи членов System.Object и System.Type:

```
public static int Main(string[] args)
{
        // Создаем класс .NET CoCalc
        CoCalc c - new CoCalc();

        // Наш COM-сервер теперь поддерживает метод System.Object.ToString()
        Console.WriteLine("-> CoCalc to string: {0}", c.ToString());

        // Извлекаем информацию о типе
        Type t = c.GetType();
        Console.WriteLine("->COMClass? : {0}", t.IsCOMObject);
        Console.WriteLine("-> Full name? : {0}", t.FullName);
```

```
        Console.WriteLine("-> CLSID? : {0}", t.GUID.ToString());
        Console.WriteLine("-> Is it a interface? : {0}", t.IsInterface);

        return 0;
}
```

Результат выполнения нашей программы представлен на рис. 12.14.

Рис. 12.14. Получение информации о COM-сервере

Все. К этому моменту мы уже умеем организовывать взаимодействие между клиентом .NET и COM-сервером, используя раннее и позднее связывание. Однако многие важные вещи еще остались нерассмотренными. В следующих разделах мы более глубоко изучим процесс представления COM-сервера как типа .NET и познакомимся с некоторыми особенностями применения утилиты tlbimp.exe.

Создаем COM-сервер при помощи ATL

Будем считать, что наши упражнения с созданием COM-сервера в Visual Basic были всего лишь разминкой. Чтобы действительно осознать процесс взаимодействия COM и .NET, мы создадим еще один COM-сервер — на этот раз при помощи ATL. Он потребуется нам для того, чтобы мы могли поработать с IDL вручную, разобраться, как будут отображаться в .NET такие элементы из мира COM, как SAFEARRAY, BSTR, перечисления, соклассы и интерфейсы. Однако я сразу должен оговориться, что эта книга — не руководство по ATL, хотя в ближайших разделах нам придется шаг за шагом создать COM-сервер с его помощью. Если вам нужна дополнительная информация по ATL (и классическому COM), рекомендую обратиться к соответствующей литературе (в том числе к моей книге *Developer's Workshop to COM and ATL 3.0*).

Итак, приступим к процессу создания COM-сервера с помощью ATL. Нам потребуется открыть Visual C++ 6.0 и воспользоваться шаблоном ATL (в принципе, можно было воспользоваться ATL 4.0 и Visual Studio.NET, однако нам важно показать взаимодействие с унаследованными приложениями, поэтому мы создадим сервер именно в ATL 3.0 и Visual Studio 6.0). Мы назовем новый проект ClassicATLCOMServer. В меню Insert (Вставка) откроем ATL Object Wizard и добавим новый Simple Object (Простой объект) с именем CoCar. Далее при помощи вкладки Names (Имена) переименуем исходный ([default]) интерфейс в ICar (рис. 12.15). Следующее, что мы должны сделать, — открыть вкладку Attributes (Атрибуты) и выбрать поддержку ISupportErrorInfo и точек соединения COM, поскольку создаваемый нами соклacc будет уметь посылать сообщения об ошибках и событиях клиенту .NET (рис. 12.16).

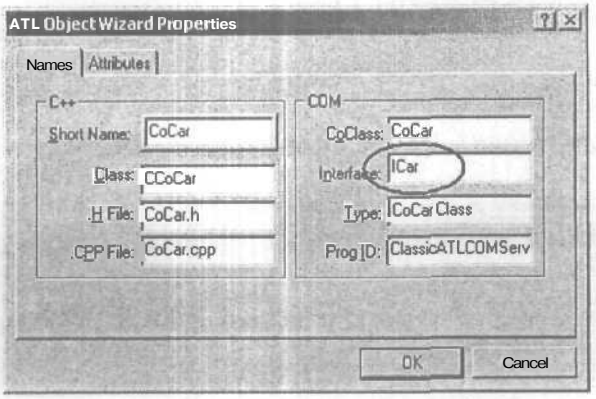


Рис. 12.15. Переименовываем интерфейс

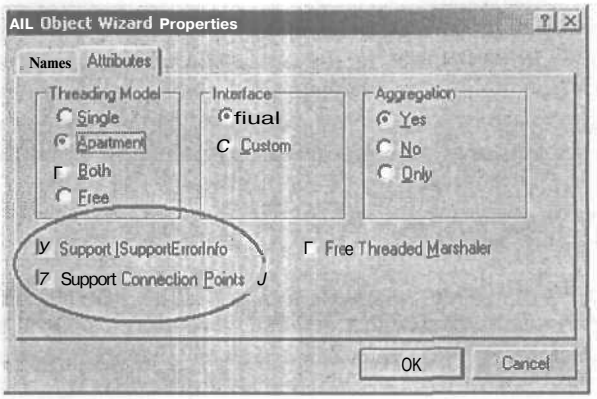


Рис. 12.16. Устанавливаем поддержку для ошибок и событий COM

Кроме того, убедимся, что мы выбрали интерфейс [dual] вместо предлагаемого по умолчанию Custom. Все остальные параметры должны остаться такими, какими они были установлены по умолчанию. Нажмем OK и приступим к работе.

Добавление методов в интерфейс по умолчанию

Первое, что мы должны сделать, создавая наш COM-сервер, — добавить несколько исходных членов в интерфейс по умолчанию ([default]), который у нас называется ICar. Щелкнем правой кнопкой мыши на значке интерфейса COM в Class View и воспользуемся мастером Add Method (Добавить метод) для добавления двух методов. Первый метод SpeedUp() будет принимать единственный параметр int (см рис. 12.17).

Второй метод — GetCurSpeed() — возвращает указатель int (помеченный как [out, retval]). После добавления обоих методов определение интерфейса в IDL будет выглядеть следующим образом:

```
[ object, uuid(A8E01A32-0300-402A-B1EC-ADCD2DC526B4), dual, helpstring"ICar interface"),
                                                    pointer_default(unique) ]
```

```
interface ICar : IDispatch
{
        [id(1), helpstring("method SpeedUp")]
        HRESULT SpeedUp([in] int delta);

        [id(2), helpString("method GetCurSpeed")]
        HRESULT GetCurSpeed([out, retval] int* currSp);
};
```

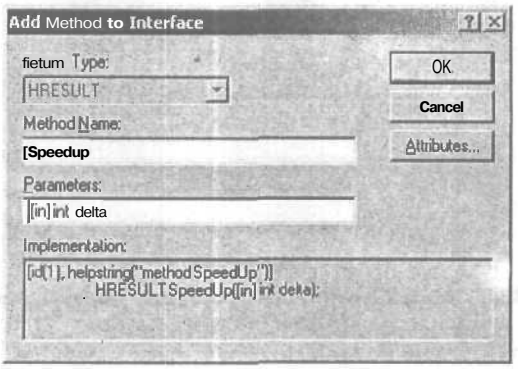


Рис. 12.17. Определяем параметры при помощи атрибутов IDL

Реализовать этот интерфейс в соклассе в соответствии с приведенным определением очень просто. Вначале мы добавим переменную типа i nt (она будет называться curSpeed), инициализируем ее в конструкторе, установив значение, равное О, а затем реализуем каждый из двух методов интерфейса следующим образом:

```
STDMETHOD CCoCar::SpeedUp (int delta)
{
        // Добавляем delta к текущей скорости
        curSpeed += delta;
        return S_OK;
}

STDMETHODIMP CCoCar::GetCurSpeed(int *currSp)
 {
        // Возвращаем текущую скорость
        *currSp = curSpeed;
        return S_OK;
}
```

Чтобы убедиться, что ошибок у нас нет, откомпилируем наш сервер.

Генерация события COM

Следующее, что мы должны сделать, — настроить наш сокласс CoCar таким образом, чтобы он мог генерировать событие COM. Первое, что мы должны при этом сделать — добавить метод в исходящий интерфейс, который будет представлять методы, вызываемые соклассом через приемник клиента (исходящий интерфейс — это интерфейс, определенный на COM-сервере, но реализованный на клиенте). В ATL исходящие интерфейсы выводятся на самом верху Class View и все они помечены префиксом "_" и суффиксом "Events". Щелкнем правой кнопкой на исхо-

дящем интерфейсе и откроем New Method Wizard. Добавим событие с именем Exploded, при возникновении которого клиенту будет передаваться параметр типа BSTR (рис. 12.18).

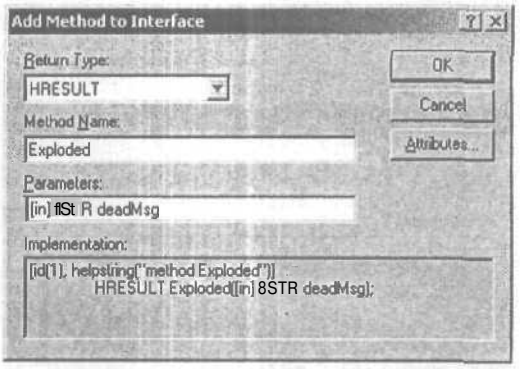


Рис. 12.18. Создаем событие для исходящего [source] интерфейса

Вот код для этого интерфейса в IDL:

```
[uuid(E88DA278-AD04-407F-9BBB-D8C00AFE7984), helpstring("_ICarEvents Interface") ]
dispinterface _ICarEvents
{
        properties:
        methods:
        [id(1), helpstring("method Exploded")]
        HRESULT Exploded([in] BSTR deadMsg );
}
```

После добавления этого события в исходящий интерфейс откомпилируем проект для того, чтобы обновить информацию о типах COM.

Следующая наша задача — добавить прокси для этого события, который будет использован средой ATL для передачи информации о возникновении события Exploded любому подключенному к COM-серверу клиенту. Щелкнем правой кнопкой мыши на классе CoCar в окне Class View и выберем в контекстном меню Implement Connection Point (Реализовать точку соединения). В открывшемся диалоговом окне выберем имя интерфейса событий (_CarEvents) и нажмем OK.

В наш сокласс CoCar будет внесено сразу несколько изменений. Во-первых, мы сможем обнаружить, что в CONNECTION_MAP (в заголовочном файле CoCar) будет внесена новая запись. Для чего нужна эта запись — нам сейчас не очень интересно, важнее исправить ошибку, которая в этом месте иногда появляется (в результате ошибки в мастере). Иногда параметр для макроса CONNECTION_POINT_ENTRY генерируется без необходимого префикса D. В этой ситуации ошибку необходимо будет исправить вручную (если у нас этой ошибки нет, нет и необходимости что-либо исправлять):

```
BEGIN_CONNECTION_POINT_MAP(CCoCar)
        // Здесь мастер может забыть префикс D
        // CONNECTION_POINT_ENTRY(IID_ICarEvents)      // Ошибка
        CONNECTION_POINT_ENTRY(DIID_ICarEvents)        // Правильно
END_CONNECTION_POINT_MAP()
```

Второе важное изменение, которое будет внесено, — в цепь наследования для CoCar будет добавлен новый класс (CProxy_ICarEvents). Этот класс реализует метод Fire_Exploded(), который и занимается всей работой по передаче информации о событии всем подключенным в настоящий момент клиентам.

Чтобы создать механизм, который будет заставлять событие Exploded срабатывать, мы добавим две переменные — maxSpeed и dead (не забудем присвоить им исходные значения в конструкторе). Переменная maxSpeed (тип данных int) будет представлять максимально допустимую скорость нашего транспортного средства. Переменная dead (тип данных bool) будет использована для хранения информации о текущем состоянии автомобиля — взорвался он уже или нет. После того как эти переменные будут добавлены в заголовочный файл, изменим метод SpeedUp() следующим образом:

```
STDMENTODIMP CCoCar::SpeedUp(int delta)
{
        // Добавляем значение delta и проверяем, не пришло ли время сработать событию
        curSpeed += delta:

        // Если автонобиль еще не взорвался, а скорость превысила максимально
        // допустимую...
        if(curSpeed >= maxSpeed && |dead)
        {
                // Должно сработать событие и установиться состояние 'dead'
                CComBSTR msg("You are toast...");

                Fire_Exploded(msg.Detach());
                curSpeed = maxSpeed:
                dead - true;
        }
        return S_OK;
}
```

Генерация ошибки COM

Вспомним, что при создании класса CoCar мы добавили поддержку протокола ошибок COM. Для наших целей нам сейчас совершенно необязательно углубляться в объекты ошибок COM на низком уровне. Все, что нам сейчас необходимо знать, — то, что нам потребуется вызвать унаследованный метод Error(), для того чтобы сообщить об ошибке в среде ATL. Продемонстрируем это на примере. Давайте внесем изменения в метод GetCurSpeed() таким образом, чтобы при попытке пользователя получить информацию о скорости автомобиля, который уже «мертв», ему выдавалось сообщение об ошибке (то есть возвращался объект ошибки COM):

```
STDMETHODIMP CCoCar::GetCurSpeed(int *currSp)
{
        // Генерируем ошибку, если автомобиль "мертв"
        if(!dead)
        {
                *currSp = curSpeed:
                return S_OK;
        }
        else
        {
                *currSp = 0:
                Error("Sorry, this car has met it's maker");
```

```
            return E_FAIL;
        }
    }
```

К этому моменту в нашем распоряжении есть сокласс COM, который поддерживает единственный пользовательский интерфейс (он же интерфейс по умолчанию — [default]). Кроме того, наш сокласс может передавать клиентам информацию о событиях и ошибках. Откомпилируем сокласс еще раз, чтобы убедиться в отсутствии опечаток.

Представление внутренних подобъектов и применение SAFEARRAY

Чтобы сокласс полностью отвечал нашим требованиям, нам осталось добавить в него еще два элемента. Первый элемент — это внутренний объект, который будет называться CoEngine и представлять двигатель нашего автомобиля. Чтобы создать его, вставим при помощи ATL Object Wizard новый ATL Simple Object, изменим имя интерфейса [default] на IEngine и изменим на вкладке Attributes (Атрибуты) тип интерфейса на [dual].

Затем используем мастер Add Method для добавления в интерфейс IEngine единственного метода, который будет называться GetCylinders(). Этот метод будет возвращать массив COM SAFEARRAY переменных BSTR, которые будут представлять прозвища для каждого цилиндра двигателя. (Да, я согласен, что прозвища цилиндрам даются не так часто, но это позволит нам возвращать массив строковых значений клиенту .NET.) Код IDL для интерфейса IEngine будет выглядеть следующим образом:

```
[object, uuid(23D2BB87-A8F8-4301-DED5-9D0CA77AE403), dual, helpstring("IEngine
                                                Interface"), pointer_default(unique) ]
Interface IEngine : IDispatch
{
        [id(1), helpstring("method GetCylinders")]
        HRESULT GetCylinders([out, retval] VARIANT* arCylinders);
};
```

А вот так может выглядеть реализация метода GetCylinders():

```
STDMETHODIMP CCoEngine::GetCylinders(VARIANT *arCylinders)
{
        VariantInit(arCylinders);
        // Массив строковых значений
        arCylinders->vt = VT_ARRAY | VT_BSTR;

        // Создаем массив
        SAFEARRAY *pSA;
        SAFEARRAYBOUND bounds = {4, 0};
        pSA = SafeArrayCreate(VT_BSTR, 1, &bounds);

        // Заполняем массив
        BSTR *theStrings;
        SafeArrayAccessData(pSA, (void**)&theStrings);
                theStrings[0] = SysAllocString(L"Grinder");
                theStrings[1] = SysAllocString(L"Oily");
                theStrings[2] = SysAllocString(L"Thumper");
                the$trings[3] = SysAllocString(L"Crusher");
```

```
    SafeArrayUnaccessData(pSA);

    // Возвращаем массив
    arCylinders->parray = pSA;

    return S_OK;
}
```

Если вы никогда не работали с COM SAFEARRAY вручную, то этот код, скорее всего, вызовет у вас недоумение. Попробуем вкратце объяснить, что к чему. SAFEARRAY — это самоописываемый массив [oleautomation]-совместимых типов данных, для которого задаются верхняя и нижняя границы тех элементов, которые могут быть в него помещены. Объекты SAFEARRAY создаются, заполняются и управляются при помощи набора библиотечных функций COM (как это было сделано в предыдущем коде). Однако нам опять-таки совершенно незачем углубляться во все эти подробности. Главное, что необходимо вынести из всего сказанного, — то, что метод GetCylinders() может быть использован внешними клиентами для получения массива строковых объектов COM (BSTR).

Наша следующая задача — обеспечить возможность клиентам COM получать ссылки на интерфейс IEngine из типа CoCar. Для этого мы будем использовать стандартный механизм COM для включения одного типа в другой. Внешний класс CoCar будет обеспечивать доступ к внутреннему классу CoEngine при помощи следующего дополнительного метода интерфейса ICar:

```
// Вспомните! Возврат указателя на интерфейс требует двойного преобразования. Также
// определение интерфейса IEngine должно быть размещено над определением интерфейса
// Icar, так. чтобы компилятор MIDL мог "увидеть" определение интерфейса
interface ICar : IDispatch
[
    ...
    [id(3>, helpString("method GetEngine")]
    HRESULT GetEngine([out, retval] IEngine** pEngine);
};
```

Реализация метода GetEngine() требует применения некоторых типов ATL. На случай, если они вам не очень знакомы, просто скажем, что объект CoEngine создается при помощи статического метода CComObject<>::CreateInstance(). После этого производится запрос к интерфейсу IEngine и возврат его клиенту:

```
STDMETHODIMP CCoCar::GetEngine(IEngine **pEngine)
{
    // Создаем CoEngine и возвращаем интерфейс IEngine клиенту
    CComObject<CCoEngine>*pEng;
    CComOnject<CCoEngine>::CreateInstance(&pEng);
    pEng->QueryInterface(IID_IEngine, (void**)pEngine);

    return S_OK;
}
```

Если мы захотим запретить создание объекта вложенного класса, достаточно добавить атрибут [noncreatable] в код IDL для CoEngine. Этот атрибут воспринимается множеством языков программирования (в том числе Visual Basic 6.0) как запрет создания пользователем объекта этого класса напрямую. Если пользователь все-таки произведет такую попытку, он получит сообщение об ошибке компилятора. Выглядеть все это может так:

```
[ uuid(32C07E17-F966-4EFD-B301-9729FE2D60B5), helpstring("CoEngine Class"),
                                                        noncreatable ]
coclass CoEngine
{
        [default] interface IEngine;
};
```

Кроме того, внесем изменения в ATL OBJECT_MAP, пометив CoEngine как NON_CREATABLE. Это запретит внешним клиентам напрямую создавать объекты этого класса:

```
BEGIN_OBJECT_MAP(ObjectMap)
        OBJECT_ENTRY(CLSID_CoCar, CCoCar)
        OBJECT_ENTRY_NON_CREATABLE(CCoEngine)
END_OBJECT_MAP()
```

Последний штрих: создаем перечисление IDL

Перед тем как мы приступим к созданию прокси-сборки для нашего COM-сервера, осталось внести в его определение последний штрих. Откроем файл IDL нашего проекта и добавим в него перечисление COM, которое будет называться CarType. Оно должно быть размещено непосредственно после выражений IMPORT в верхней части файла:

```
// Это перечисление COH будет использовано для идентификации типа автомобиля
typedef enum CarType {Jetta, BMW, Ford, Colt} CarType;
```

Чтобы внешние клиенты смогли обращаться к этому перечислению, добавим в интерфейс ICar еще один метод, который будет называться GetCarType():

```
interface ICar : IDispatch
{
        [id(1), helpstring("method SpeedUp")]
        HRESULT SpeedUp([in] int delta);

        [id(2), helpstring("method GetCurSpeed")]
        HRESULT GetCurSpeed([out, retval] int* currSp);

        [id(3), helpstring("method GetEngine)]
        HRESULT GetEngine([out, retval] IEngine** pEngine);

        [1d(4), helpstring("method GetCarType")]
        HRESULT GetCarType([out, retval] CarType* ct);
};
```

Реализация метода GetCarType() создает и устанавливает значение перечисления CarType для использования клиентом COM:

```
STDMETHODIMP CCoCar::GetCarType(CarType *ct)
{
        *ct = Colt; // Или что-нибудь другое...
        return S_OK;
}
```

Готов поспорить, что вы вряд ли ожидали встретить столько информации по ATL в книге, посвященной C#. В свое оправдание могу сказать только, что все это нам понадобится для рассмотрения вопросов взаимодействия .NET и COM. В большинстве случаев COM-серверы выполняют гораздо более сложные дей-

ствия, чем сложение двух объектов, вот почему нам потребовался COM-сервер с поддержкой протокола ошибок, точек соединения, вложенным классом, перечислением, который работает с такими распространенными в мире COM типами, как BSTR и SAFEARRAY.

Теперь, когда работа над COM-сервером завершена, мы создадим промежуточную сборку и рассмотрим, как более сложные возможности COM будут отображаться в .NET. Однако перед этим мы вначале опробуем наш класс CoCar в действии, подключив к нему традиционного клиента на Visual Basic 6.0.

Код приложения ClassicATLCOMServer можно найти в подкаталоге Chapter 12.

Клиент COM-сервера в Visual Basic 6.0

Интерфейс клиента Visual Basic 6,0 представлен на рис. 12.19.

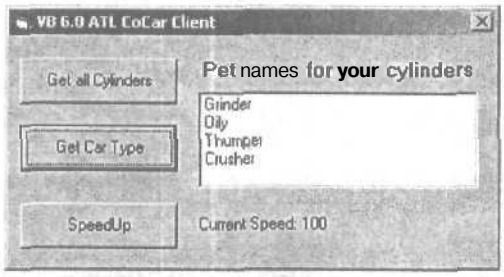


Рис. 12.19. Интерфейс клиента Visual Basic

При загрузке формы создается объект класса Car (который объявлен с ключевым словом WithEvents, чтобы можно было реализовать реакцию на входящее событие Exploded). Обработчик события Exploded всего-навсего выводит сообщение, посылаемое погибающим автомобилем;

```
Private Sub myCar_Exploded(ByVal deadMsg As String)
        MsgBox deadMsg, . "Message from CoCar!"
End Sub
```

При нажатии на кнопку Speedup происходит, конечно, увеличение скорости автомобиля. Вспомним, что как только скорость автомобиля достигнет максимума, COM-сервер CoCar отправит сообщение о событии Exploded. Если пользователь попробует ускорить уже несуществующий автомобиль, он получит объект COM-ошибки, которую Visual Basic перехватит при помощи конструкции On Error GoTo:

```
Private Sub btnSpeedUp_Click()
On Error GoTo OOPS

        myCar.SpeedUp 50
        Label2.Caption = Current Speed:  " & myCar.GetCurSpeed

Exit Sub

OOPS:
        MsgBox Err.Description, . "Error from car!"
```

```
Resume Next
End Sub
```

При нажатии на кнопку Get all Cylinders будет получена от CoCar ссылка на интерфейс IEngine, а после этого будет вызван метод GetCylinders():

```
Private Sub btnGetCylinders_Click()

        ' Прежде всего нам необходимо получить двигатель
        Dim q as CoEngine
        Set q = myCar.GetEngine

        ' Теперь получаем цилиндры
        01m strs As Variant
        strs = q.GetCylinders

        ' Теперь получаем все прозвища из SAFEARRAY
        ' и помещаем их в ListBox
        Dim upper As Integer
        Dim i As Integer
        upper = UBound(Strs)
        For i = () To Upper
              lstCylinderList.AddItem strs(i)
        Next i

End Sub
```

Код приложения VB6ATL можно найти в подкаталоге Chapter 12.

Создаем сборку и анализируем процесс преобразования

Давайте откроем командную строку и воспользуемся утилитой tlbimp.exe для создания промежуточной сборки для нашего COM-сервера ClassicATLCOMServer.dll:

```
tlbimp classicatlcomserver.dll /out:AtlServerAssembly.dll
```

Теперь загрузим промежуточную сборку в ILDasm.exe и рассмотрим типы, которые были сгенерированы автоматически (рис. 12.20). Как можно видеть, в сборке появились аналоги .NET для типов CarType, CoCar и CoEngi ne (то, во что превратились события COM-сервера, мы вскоре рассмотрим более подробно). Давайте разберемся с особенностями произведенных преобразований.

Преобразование библиотеки типов

При создании промежуточной сборки для COM-сервера (прокси-сборки) в пространстве имен .NET был создан аналог каждого типа COM (перечисления, сокласса, пользовательского интерфейса).

Первое, на что мы обратим внимание, — что атрибут [version] библиотеки типов был использован для определения версии сборки. Если мы обновим в коде IDL атрибут версии, увеличив номер версии до 9.7, примерно так:

```
[ uuid(69D8B2E2-4CC1-4414-9757-49C53620FF0C), version(9.7),
                  helpstring("ClassicATLCOMServer 1.0 Type Library") ]
library CLASSICATLCOMSERVERLib
{
        // Весь необходимый код...
}
```

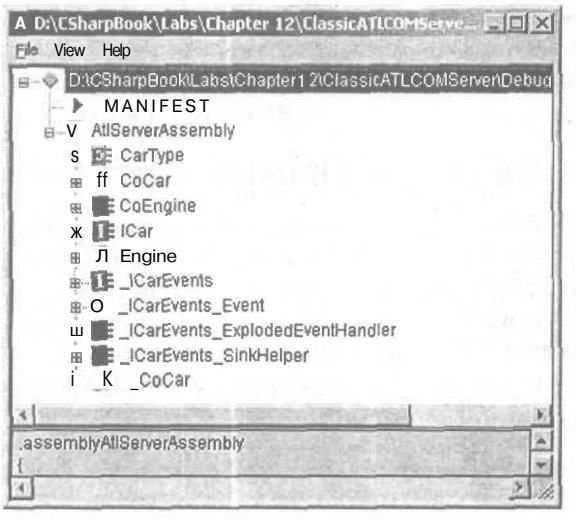


Рис. 12.20. Промежуточная сборка, созданная нами для COM-сервера

то в манифесте сборки мы сможем увидеть соответствующую информацию о номере версии сборки:

```
.assembly AtlServerAssembly
{
...
        .ver 9:7:0:0
}
```

Преобразование интерфейсов COM

Когда в сборке .NET создается аналог интерфейса COM, он описывается при помощи разнообразных атрибутов из пространства имен System.Runtime.InteropServices.

Первый атрибут, который используется для создания аналога, — атрибут GuidAttribute. Конечно же, он нужен для записи информации об идентификаторе интерфейса (IID) в соответствии с тем, как этот идентификатор записан в IDL COM-сервера.

Второй атрибут — InterfaceTypeAttribute. Он используется для хранения информации о том, как данный интерфейс был определен в коде IDL (custom, dual или dispinterface). Для этого атрибута используются значения из перечисления ComInterfaceType (табл. 12.6).

Таблица 12.6. Значения перечисления ComInterfaceType

Значение	Описание
InterfaceIsDual	Определяет, что интерфейс описан в COM-сервере как dual
InterfaceIsDispatch	Определяет, что интерфейс был описан как dispinterface
InterfaceIsUnknown	Определяет, что интерфейс был описан как производный от IUnknown

Для интерфейсов dual (как в нашем случае) атрибут InterfaceTypeAttribute в прокси-сборке не используется. Вместо этого используется другой атрибут - TypeLibTypeAttribute, в котором и хранится информация о разновидности интерфейса типа dual (например, licensed, hidden, appobject и т. д.).

Преобразование атрибутов параметров в коде IDL

В классическом COM параметры помечаются при помощи набора атрибутов IDL. Эти атрибуты определяют направление передачи заданного аргумента. Они также нужны для правильного управления памятью. В наших соклассах использовались параметры с атрибутами [in] и [out, retval]. Другие допустимые значения — [out] и [in, out]. Чтобы проиллюстрировать все возможности, предположим, что в коде IDL вашего COM-сервера присутствует определение интерфейса IParams:

```
interface IParams : IDispatch
t
        // Параметры [in] генерируются вызывающим клиентом
        [id(1)] HRESULT OnlyInParams([in] int x. [in] int y):

        // Параметры [out] генерируются вызываемым методом
        [id(2)] HRESULT OnlyOutParams([out] int* x. [out] int* y);

        // Параметры [retval] - это исходящие параметры, которым соответствуют физические
        // значения. Например, в VB это могло бы выглядеть как "ans = Retval()"
        [id(3)] HRESULT Retval([out, retval] int* answer);

        // Параметры [in, out] генерируются вызывающим клиентом, однако могут быть
        // изменены вызываемым методом
        [id(4)] HRESULT InAndOut([in, out] int* byRefParam);
}:
```

Если мы создадим для COM-сервера с таким определением интерфейса прокси-сборку при помощи утилиты tlbimp.exe, то сможем обнаружить, что каждой из приведенных разновидностей параметров подобран свой аналог .NET. Выглядеть наш интерфейс IParams в прокси-сборке, если открыть его в Object Browser, может так, как представлено на рис. 12.21,

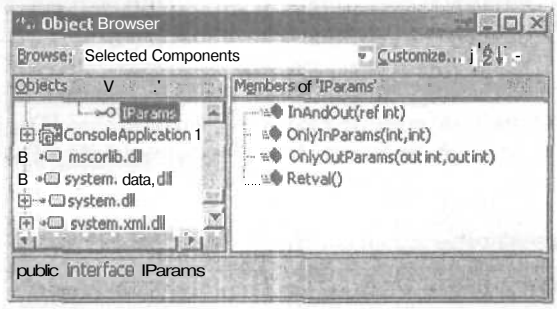


Рис. 12.21. Интерфейс IParams перенесен в прокси-сборку

Общая схема преобразования определений параметров из кода IDL в код C# может выглядеть так, как представлено в табл. 12.7.

Таблица 12.7. Преобразование атрибутов параметров IDL в ключевые слова C#

Атрибут параметра в IDL	Ключевое слово для параметра в C#	Описание
[in]	Ключевого слова не предусмотрено — это направление передачи параметра по умолчанию	Вызываемой функции передается копия данных
[out]	out	Значение генерируется вызываемой функцией и передается вызывающему клиенту
[in,out]	ref	Значение генерируется вызывающим клиентом, однако оно может быть изменено вызываемой функцией
[out, retval]	Нет	Эти параметры становятся физическими значениями, возвращаемыми функциями. При преобразовании в .NET фактически эти параметры удаляются (преобразуются в тип void)

Преобразование иерархии интерфейсов

Несмотря на то что в нашем COM-сервере нет иерархии пользовательских интерфейсов, предположим (для примера), что у нас появился еще один интерфейс ITurboEngine, производный от IEngine. Код IDL для него может выглядеть следующим образом:

```
interface ITurboEngine : IEngine
{
        HRESULT PowerBoost();
};
```

В процессе преобразования производный интерфейс будет представлен как объединение всех методов, определенных во всех базовых интерфейсах и в самом этом интерфейсе (рис. 12.22). Таким образом, если мы исследуем ITurboEngine при помощи ILDasm.exe, то мы обнаружим, что он поддерживает не только свой метод PowerBoost(), но и метод GetCylinders(), который был определен в базовом интерфейсе. Обратите также внимание, что имя базового интерфейса помечено тегом [implements].

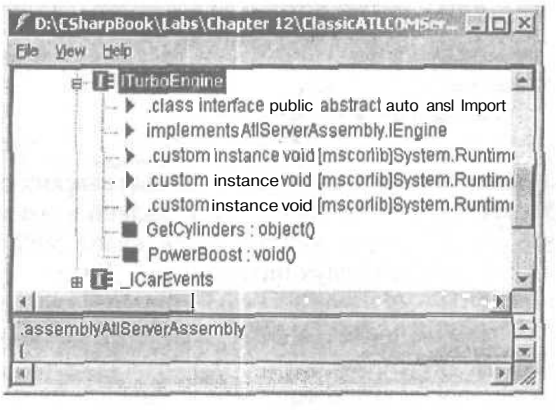


Рис. 12.22. Производные интерфейсы поддерживают все члены базовых интерфейсов

Преобразование соклассов и свойств COM

Как мы уже могли убедиться на примере с COM-сервером в Visual Basic, при создании прокси-сборки для COM-сервера при помощи утилиты tLbimp.exe специальные классы .NET создаются не только для каждого интерфейса, но и для всего сокласса в целом. Поэтому в прокси-сборке у нас появился новый класс .NET с именем CoCar. Работать с ним можно двумя способами. Первый способ — создать объект сокласса и получить доступ к членам интерфейсов через этот объект:

```
// Реально мы обращаемся к члену интерфейса [default]
CoCar viper = new CoCar();
viper.SpeedUp(30);
```

Второй способ — явно запросить ссылку на интерфейс ICar и работать уже через него:

```
// Явно получаем ссылку на интерфейс ICar
CoCar viper = new CoCar0:
ICar ic = (ICar)viper;
ic.SpeedUp(30);
```

В нашем примере и класс CoCar, и класс CoEngine реализуют только по одному пользовательскому интерфейсу (которые одновременно являются интерфейсами по умолчанию). Однако предположим, что мы определили еще один интерфейс с единственным свойством COM (типа BSTR):

```
interface IDriverInfo : IDispatch
{
      [id(1), propget, helpstring("property DriverName")]
      HRESULT DriverName([out, retval] BSTR *pVal);

      [id(1), propput, helpstring("property DriverName")]
      HRESULT DriverName([in] BSTR newVal);
};
```

Также предположим, что наш класс CoCar реализует этот интерфейс. Реализация будет несложной — при помощи этого свойства можно будет устанавливать значение типа BSTR или возвращать это значение:

```
coclass CoCar
{
      [default] interface ICar;
      interface IDriverInfo;
      [default, source] dispinterface _ICarEvents;
};
```

Теперь, когда сокласс CoCar реализует два пользовательских интерфейса, интересно будет разобраться, как все это будет представлено в его аналоге в прокси-сборке .NET. Как вы, наверное, уже догадываетесь, класс .NET будет поддерживать каждый член каждого из этих двух интерфейсов. Другими словами, если мы создадим объект класса .NET CoCar, то сможем воспользоваться и методом SpeedUp(), и GetCurrentSpeed(), и GetEngine(), и GetCarType() и работать со свойством DriverName:

```
// Обратите внимание, что мы можем обратиться к свойству, определенному в IDriverInfo
// напрямую через объект класса, реализующего этот интерфейс
CoCar viper = new CoCar0:
```

```
viper.DriverName = "Fred";
Console.WriteLine(viper.DriverName);
```

Точно так же, как и раньше, мы можем обратиться к свойству DriverName, явно получив ссылку на интерфейс IDriverInfo:

```
// Устанавливаем значение свойства DrlverName и получаем это значение через ссылку
// на интерфейс IDriverInfo
IDriverInfo idi = (IDriverInfo)viper;
idi.DriverName = "Fred";
Console.WriteLine("Name of driver is: " + idi.DriverName);
```

Однако если мы откроем прокси-сборку в ILDasm.exe, то этих членов интерфейсов (например, свойства DriverName) мы в определении класса .NET CoCar не найдем (рис. 12.23).

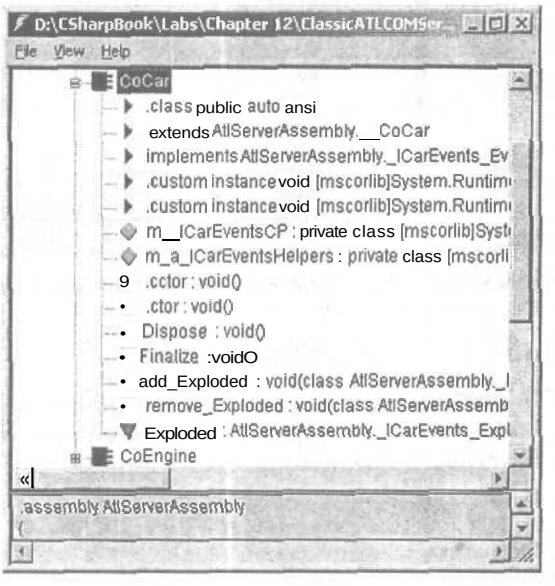


Рис. 12.23. Классы .NET не реализуют методы интерфейсов COM-серверов напрямую

Откуда же берутся эти члены? Если разобраться повнимательнее, то можно заметить, что класс .NET CoCar является производным от другого класса .NET _CoCar. И вот именно в этом базовом классе _CoCar и определены все члены интерфейсов ICar и IDriverInfo (рис. 12.24).

Если обратиться к метаданным типов, то можно обнаружить следующее определение класса _CoCar:

```
.class public auto ansi import _CoCar
       extends [mscorlib]System.Object
       implements CLASSICATLCOMSERVERLib.ICar, CLASSICATLCOMSERVERLib.IDriverInfo
{
       // Атрибут TypeLibTypeAttribute
       // Атрибут ComSourceInterfacesAttribute
       // Атрибут GuidAttribute
       // Атрибут HasDefaultInterfaceAttribute

}      // Конец определения класса _CoCar
```

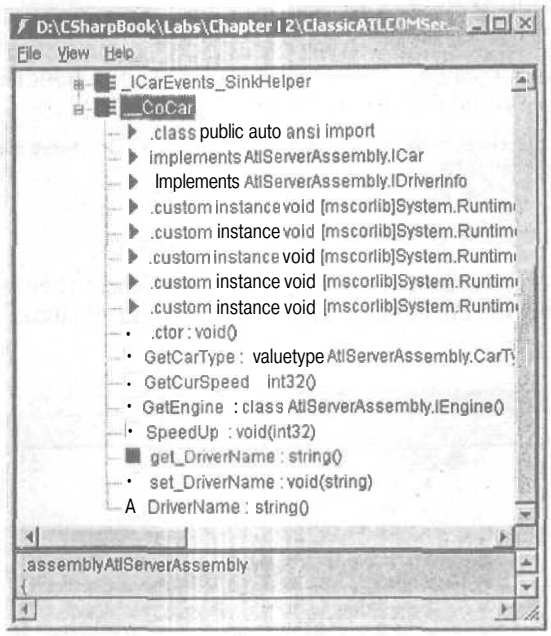


Рис. 12.24. Члены интерфейсов определяются в дополнительном базовом классе

Преобразование перечислений COM

Перечисления COM отображаются в прокси-сборке для COM-сервера в типы, производные от System.Enum. Таким образом, мы можем использовать любые из встроенных членов (к слову, очень полезных) этих типов .NET. Например:

```
public static int Ma1n(string[] args)
{
        // Вначале создаем объект «автомобиль»
        CoCar viper = new CoCar();

        // Далее получаем тип автомобиля
        CarType t = viper.GetCarType();
        Console.WriteLine("Car type: {0}", t.ToString());

        return 0;
}
```

Преобразование COM SAFEARRAY

Давайте рассмотрим на примере, как представляется тип COM SAFEARRAY в прокси-сборке для COM-сервера. Как мы помним, в интерфейсе IEngine у нас был определен единственный метод GetCylinders(), который возвращал массив объектов BSTR. Обращение к интерфейсу lEngi ne из внешних клиентов производилось при помощи метода GetEngine() объекта CoCar, Выглядело это обращение следующим образом:

```
// Прежде всего создаем объект CoCar
CoCar viper = new CoCarO;
```

```
// Далее получаем ссылку на интерфейс IEngine
IEngine e = viper.GetEngine();

// Запрашиваем массив SAFEARRAY через ссылку на интерфейс
object o = e.GetCylinders();
```

Наш массив SAFEARRAY был объявлен как массив типов VARIANT при помощи флагов VT_ARRAY и VT_BSTR:

```
STDNETHODIMP CCoEngine::GetCylinders(VARIANT *arCylinders)
{
        VariantInit(arCylinders);
        arCylinders->vt = VT_ARRAY | VT_BSTR;

        // Здесь расположены различные элементы COM...

        // Устанавливаем возвращаемое значение
        arCylinders->parray = pSA;

        return S_OK;
}
```

Для целей, для которых в мире COM используется тип VARIANT, в мире .NET используется System.Object. Учитывая, что SAFEARRAY в нашем случае заполнен строковыми значениями, мы можем вывести информацию о том, что представляет собой в нашей ситуации объект System.Object, прямо на консоль:

```
IEngine e = viper.GetEngine();
object o = e.GetCylindersO:
// o в нашем случае - объект System.String[]

Console.WriteLine("o is really this type: {0}", o);
```

Для вывода прозвища каждого из цилиндров код может быть таким:

```
// Получаем массив строковых значений
String[] cylinders = (string[])o;

// Выводим каждый элемент .
Console.WriteLine("Your cylinders are:");
froeach(string s in cylinders)
(
        Console.WriteLine("->" + s);
}
```

То, что должно получиться, представлено на рис. 12.25.

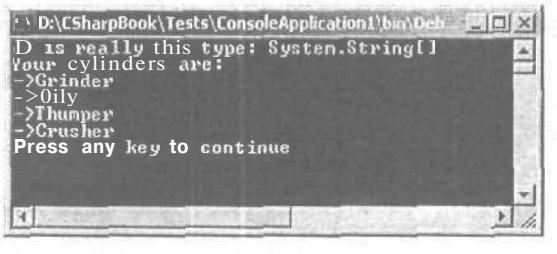


Рис. 12.25. Вывод каждого элемента SAFEARRAY на консоль

Здорово, не правда ли? Таким образом, мы можем использовать значения из COM SAFEARRAY в коде .NET без каких-либо проблем. Следующий пункт, с которым мы должны разобраться, — как организовано представление событий COM в .NET.

Перехват событий COM

Модель событий .NET в этой книге рассматривалась в главе 5. Напомним вкратце ее основные положения. Архитектура модели событий .NET основывается на делегировании логики выполнения от одной части приложения к другой. Для подобного делегирования используются типы, производные от System.MulticastDelegate. Клиент может добавлять и удалять приемники событий из внутреннего списка при помощи перегруженных операторов += и -=.

Когда утилита tlbimp.exe в процессе преобразования встречает интерфейс [source] в библиотеке типов COM-сервера, она создает набор типов .NET, которые послужат оболочкой для низкоуровневой архитектуры точек соединения COM. Можно сказать, что в прокси-сборке создаются эквиваленты .NET для событий COM. Например, как мы помним, в нашем соклассе CoCar был определен следующий исходящий интерфейс:

```
dispinterface _ICarEvents
{
        properties:
        methods:
        [id(1), helpstring("method Exploded")]
        HRESULT Exploded([in] BSTR deadMsg );
};
```

Встретив такой интерфейс, утилита tlbimp.exe создала набор типов для точного представления системы событий COM-сервера (архитектуры точек соединения) в системе событий .NET. Эти типы приведены в табл. 12.8.

Таблица 12.8. Соответствия сгенерированных элементов системы событий .NET элементам системы событий COM

Сгенерированный тип (для интерфейса _CarEvents [source])	Описание
_ICarEvents	Интерфейс .NET — аналог исходящего интерфейса COM-сервера. Обычно напрямую не используется
_ICarEvents_Event	Интерфейс .NET, определяющий члены для добавления и удаления методов из встроенного списка System.MulticastDelegate. Обычно также напрямую не используется
_ICarEvents_ExplodedEventHandler	Делегат .NET (тип, производный от System.MulticastDelegate). Обработчик события должен обязательно возвращать значение типа int. Этот тип соответствует исходному событию COM
_ICarEvents_SinkHelper	Этот сгенерированный класс реализует исходящий интерфейс в объекте-приемнике .NET. Этот класс присваивает значение cookie, сгенерированное типом COM переменной m_dwCookie. Кроме того, в этом классе предусмотрена внутренняя переменная m_ExplodedDelegate, представляющая исходящий интерфейс (_ICarEvents_ExplodedEventHandler)

Помимо сгенерированных типов, представленных в табл. 12.8, конечно, для системы событий вносятся изменения и в определение класса .NET CoCar. В определении этого класса мы сможем обнаружить новое событие Exploded:

```
.class public auto ansi CoCar
       extends AtlServerAssembly._CoCar
       implements AtlServerAssembly._ICarEvents Event
{
...
} // Конец класса CoCar
```

В классе .NET CoCar предусмотрено два метода, определенные как private, которые обслуживают соединение с источником событий COM — add_X и remove X (рис. 12.26).

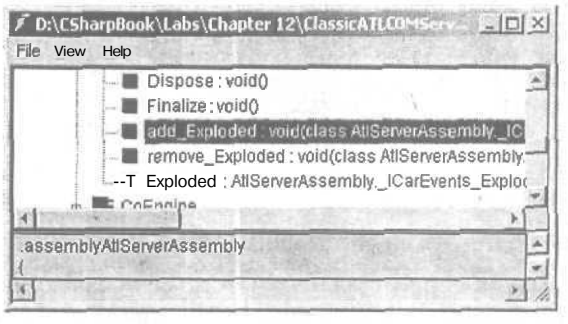


Рис. 12.26. События COM обслуживаются при помощи набора из двух функций

Если мы заглянем в код инструкций IL для метода add_Exploded, то сможем обнаружить, что при его выполнении создается новый объект _ICarEvents_SinkHelper. После этого прокси-сборка получает ссылку на интерфейс IConnectionPoint из COM-сервера CoCar, вызывает метод Advise() и кэширует возвращаемое значение cookie. Вот соответствующие инструкции в коде IL (еще раз отметим, что для нормальной работы полностью разбираться в тонкостях IL вовсе не обязательно):

```
.method public virtual instance void add_Exploded (class
            AtlServerAssembly._ICarEvents_ExplodedEventHandler A_1) cil managed
{
...
// Создается вспомогательный объект приемника
IL_0000:    newobj instance void AtlServerAssembly._ICarEvents_SinkHelper::.ctor()
IL_0005:    stloc.0
IL_0006:    ldc.i4.0
IL_0007:    stloc.1
IL_0008:    ldarg  0
IL_000c:    ldfld  class

// Получаем ссылку на интерфейс IConnectionPoint
[mscorlib]System.Runtime.InteropServices.UCOMIConnectionPoint
                        AtlServerAssembly.CoCar::m_ICarEventsCP
IL_0011:    ldloc.0
IL_0012:    castclass    [mscorlib]System.Object
IL_0017:    ldloca.s     V_1
```

```
IL_0019:      callvirt      instance void

// Вызываем метод IConnectionPoint::Advise()
[mscorlib]System.Runtime.InteropServices.UCOMIConnectionPoint::Advise(object, int32&)
IL_001e:      ldloc.0
IL_001f:      ldloc.1

// Сохраняем значение cookie, возвращенное методом Advise()
IL_0020:    stfld  int32 AtlServerAssembly._ICarEvents_SinkHelper::m_dwCookie
IL_0025: ldloc.0
IL_0026: ldarg       A_I

// Добавляем SinkHelper в делегат
IL_002a:     stfld  class AtlServerAssembly._ICarEvents_ExplodedEventHandler
                    AtlServerAssembly._ICarEvents_SinkHelper::m_ExplodedDelegate
IL_002f:     ldarg  0
IL_0033:     ldfld  class [mscorlib]System.Collections.ArrayList
                    AtlServerAssembly.CoCar::m_a_ICarEventsHelpers
IL_0038:     ldloc.0
IL_0039:     castclass     [mscorlib]System.Object
IL_003e:     callvirt      instance int32
                    [mscorlib]System.ollections.ArrayList::Add(object)
IL_0043:     pop
IL_0044:     ret
}  // Конец метопа CoCar::add_Exploded
```

Захват события COM прокси-сборкой .NET

Теперь, когда мы получили представление о том, какие аналоги элементам системы событий COM создаются в прокси-сборке .NET, разобраться в том, как сборка .NET реагирует на событие COM-сервера, совсем просто. Этот процесс почти идентичен процессу работы с делегатами .NET:

```
public class CoCarClient
{
      // Этот метод будет вызван при возникновении события на COM-сервере. Он должен
      // обязательно возвращать значение типа int!
      public static int Exploded Handler(String msg)
      {
            Console.WriteLine("\nCar says: (COM Events)\n->" + msg + "\n");

            return 0:
      }
      public static int Main(str1ng[] args)
      {
            CoCar viper = new CoCar();

            // Устанавливаем делегат для события Exploded
            viper.Exploded += new _ICarEvents_ExplodedEventHandler(ExplodedHandler);

            // Эти действия мы производим, чтобы COM-сервер сгенерировал событие
            for(int 1=0: i < 5; i++)
            {
                  try
                  {
                        viper.SpeedUp(50);
                        Console.WriteLine("->Curr speed is: " +
                                          viper.GetCurSpeed());
                  }
                  catch(Exception ex)
```

```
            {
                Console.WriteLine("->COM error! " + ex.Message + "\n");
            }
        }
    }
}
```

Можно сказать, что все, что нам нужно, — это установить делегат для события Exploded:

```
viper.Exploded += new _ICarEvents_ExplodedEventHandler(ExplodedHandler);
```

При возникновении события Exploded на COM-сервере (это происходит, когда скорость превышает предельно допустимую) обработчик события ExplodedHandler запускается автоматически.

Обработка ошибки COM

Обратите внимание, что мы поместили код для ускорения автомобиля в блок try/catch. Если пользователь попытается ускорить уже взорвавшийся автомобиль, наш COM-сервер вернет объект ошибки COM. Этот объект будет преобразован прокси-сборкой в исключение .NET.

Полный код клиента C#

В принципе, мы уже разобрались во всех моментах, которые возникают при создании клиента .NET, обращающегося к COM-серверу через прокси-сборку. Ниже все это сведено воедино. Вот полный код клиента C# для нашего COM-сервера:

```
using System;
// Пространство имен прокси-сборки
using AtlServerAssembly;
using System.Reflection;

public class CoCarClient
{
    public static int ExplodedHandler(String msg)
    {
        Console.WriteLine("\nCar says: (COM Events)\n->" + msg + "\n");
        return 0;
    }

    public static int Main(string[] args)
    {
        // Начинаем с создания объекта «автонобиль»
        CoCar viper = new CoCar();

        // Устанавливаем делегат для события Exploded
        viper.Exploded += new _ICarEvents_ExplodedEventHandler(ExplodedHandler);

        // Задаем имя водителя и получаем о нем информацию
        viper.DriverName = "Fred";
        Console.WriteLine("Driver is named: (COM Property)\n->" +
                                            viper.DriverName + "\n");

        // Выводим информацию о типе автонобиля
        CarType t = viper.GetCarType();
```

```
            Console.WriteLine("Car type is: (COM enum)\n->" + t.ToString() + "\n");

            // А теперь обращаемся к двигателю и цилиндрам
            IEngine e = viper.GetEngine();
            object o = e.GetCylinders();

            // Распаковываем ссылку на объект в массив строковых значений
            String[] cylinders = (string[])o;

            // Выводим информацию о каждом из цилиндров
            Console.WriteLine("Your cylinders are: ");
            foreach(string s in cylinders)
            {
                  Console.WriteLine("->" + s);
            }

            // А теперь разгоняем автомобиль, чтобы он взорвался
            for(int i=0; i < 5; i++)
            {
                  try
                  {
                        viper.SpeedUp(50);
                        Console.WriteLine("->Curr speed is: " +
                                          viper.GetCurSpeed());
                  }
                  // Перехватываем ошибку COM
                  catch(Exception ex)
                  {
                        Console.WriteLine("->COM error! " + ex.Message + "\n");
                  }
            }
      return 0;
      }
}
```

Результат работы нашего клиента представлен на рис. 12.27.

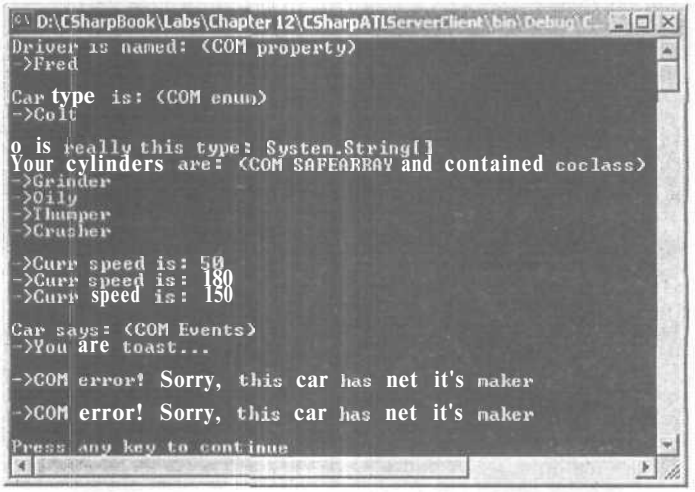


Рис. 12.27. Обращение из клиента C# к серверу ATL

Обращение клиента COM к сборке .NET

В этой части этой главы рассматривается противоположная задача: как сделать так, чтобы класс COM мог обращаться к сборке .NET. Такая ситуация встречается реже, чем обратная, но все-таки встречается.

Общий принцип решения этой задачи очевиден: мы «должны обмануть» класс COM, представив сборку .NET в виде обычного программного модуля COM. Например, тип COM должен иметь возможность получать от нашей сборки ссылки на интерфейсы при помощи метода QueryInterface O, должен «управлять памятью» (в чем, как мы помним, сборки .NET совсем не нуждаются) при помощи методов AddRef() и Release(), применять протокол точек соединения COM и т. п.

Помимо клиента COM наша сборка должна также уметь обманывать среду выполнения COM. Классический COM-сервер активизируется при помощи специального программного модуля, называемого Service Control Manager (SCM). SCM, в частности, активно работает с реестром операционной системы, получая из него информацию о ProgID, CLSID, IID и прочих жизненно важных данных COM. Одна из потенциальных проблем заключается в том, что, как мы знаем, информация о сборках .NET вообще не заносится в системный реестр.

Сводя все вышесказанное воедино, можно сказать, что для создания сборки .NET, к которой сможет обращаться клиент COM, нам потребуется сделать следующее:

- зарегистрировать сборку в реестре, чтобы она могла быть обнаружена модулем SCM;
- создать библиотеку типов COM (*.tlb), основанную на метаданных .NET, через которую клиенты COM смогут взаимодействовать с типами .NET;
- поместить сборку в тот же каталог, в котором расположен клиентCOM, или установить сборку в GAC.

Конечно же, в .NET предусмотрено средство, которое позволяет решать эту проблему автоматически. Однако прежде чем приступить к работе с этим средством, мы исследуем теоретические аспекты организации взаимодействия клиентов COM со сборками .NET посредством CCW.

Роль CCW

Для обеспечения доступа клиента COM к типу .NET используется специальный промежуточный уровень, называемый CCW (COM Callable Wrapper, оболочка для COM с возможностью вызова). Роль CCW представлена на рис. 12.28.

В CCW предусмотрены средства управления оперативной памятью — то есть отслеживание создаваемых объектов и удаление их из оперативной памяти, когда потребность в них отпадет. Это — обязательное условие, поскольку модули CCW являются настоящими типами COM и поэтому они должны уметь правильно реагировать на вызовы методов AddRefO и Release(), поступающие от клиентов. Когда клиент COM полностью освободит все ресурсы, модуль CCW автоматически освободит ссылки на используемые им типы .NET, предоставляя возможность сборщику мусора заняться своим делом.

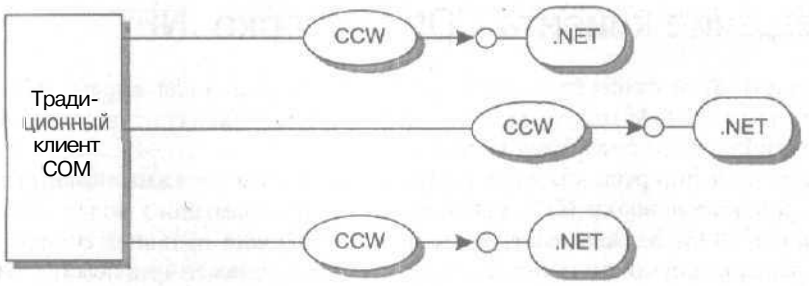


Рис. 1.2.28. Обращение клиентов COM к типам .NET при помощи CCW

Модули CCW автоматически реализуют множество служебных интерфейсов COM, чтобы создать у клиентов COM полную иллюзию, что они имеют дело с настоящим соклассом. Эти стандартные интерфейсы представлены в табл. 12.9. Помимо них модули CCW, конечно же, реализуют и пользовательские интерфейсы типов .NET, включая так называемый интерфейс класса, о котором будет рассказано чуть ниже.

Таблица 12.9. Служебные интерфейсы COM, поддержка которых автоматически обеспечивается в модулях CCW

Служебный интерфейс COM	Описание
IConnectionPointContainer IConnectionPoint	Если в типе .NET определены какие-либо события, то эти интерфейсы используются для реализации точек соединения COM
IEnumVariant	Этот интерфейс реализуется, если тип .NET поддерживает интерфейс IEnumerable
ISupportErrorInfo IErrorInfo	Эти интерфейсы позволяют передавать объекты ошибок COM
ITypeInfo IProvideClassInfo	Эти интерфейсы разрешают клиенту COM обращаться к информации типов COM модуля CCW. В действительности клиенту COM передается информация, сгенерированная на основе метаданных .NET
IUnknown IDispatch IDispatchEx	Эти важнейшие интерфейсы COM обеспечивают возможность применения раннего и позднего связывания к типам .NET. Интерфейс IDispatchEx реализуется модулем CCW в том случае, если тип .NET реализует интерфейс IExpando

Понятие интерфейса класса

В классическом COM единственный способ, при помощи которого клиент COM может взаимодействовать с объектом COM, — это получение ссылки на интерфейс. Однако некоторые языки программирования, на которых можно работать с COM (в том числе, например, Visual Basic 6.0), скрывают эту особенность от программистов — в основном для простоты восприятия. Visual Basic автоматически создает интерфейс по умолчанию ([default]) для каждого соклacca в двоичном модуле COM. В этот интерфейс помешаются все члены, объявленные как Public. Несмотря на то что клиенты Visual Basic внешне работают со ссылкой на объект, на самом деле они работают со ссылкой на интерфейс:

```
' Visual Basic прячет от программиста интерфейс [default]
' Получаем ссылку на интерфейс [default] _MyComClass
Dim o as New MyComClass
' На саном деле вызываем метод _MyComClass->Hello()
o.Hello
```

В отличие от типов COM типы .NET в принципе могут вообще обойтись без интерфейсов. Например, мы можем создать законченное решение приличного размера, используя только ссылки на объекты. Однако, если нам потребуется обеспечить клиентам COM возможность обращаться к сборке .NET, нам никуда не уйти от того факта, что клиенты COM смогут работать только через ссылки на интерфейс. Поэтому в промежуточном модуле CCW создается специальный интерфейс класса, в который помещаются все свойства, методы, поля и события, которые определены в типе .NET как public. Можно сказать, что в модулях CCW используется тот же подход, что и в Visual Basic 6.0.

Определяем интерфейс класса

Для того чтобы вмешаться в процесс автоматического создания интерфейса класса, можно использовать атрибут ClassInterface. Этот атрибут не является обязательным, но использовать его приходится достаточно часто. По умолчанию для интерфейса класса в модуле CCW выбирается тип диспинтерфейса (то есть интерфейса, производного от IDispatch). Таким образом, если оставить все по умолчанию, всем клиентам COM, которые хотят вызывать методы из сборки .NET, придется использовать позднее связывание! Для того чтобы изменить тип создаваемого интерфейса класса, и используется атрибут ClassInterface. Для него используются значения из перечисления ClassInterfaceType (табл. 12.10).

Таблица 12.10. Значения перечисления ClassInterfaceType

Значение	Описание
AutoDispatch	Определяет, что для интерфейса класса будет выбран тип диспинтерфейса (то есть интерфейса, производного от IDispatch)
AutoDual	Определяет, что для интерфейса класса будет выбран тип dual
None	Интерфейс класса вообще создаваться не будет

В нашем примере мы выберем значение ClassInterfaceType.AutoDual. Это позволит клиентам, применяющим позднее связывание (например, клиентам на VBScript), получать доступ к методам Add() и Subtract() через ссылку на IDispatch, в то время как клиенты, применяющие раннее связывание (на обычном Visual Basic и C++), смогут использовать ссылку на интерфейс класса (он будет называться _CSharpCalc). Точно так же, как в Visual Basic, имя интерфейса класса всегда состоит из префикса в виде символа подчеркивания и имени самого класса.

Создание типа .NET

Мы обсуждаем вопросы, связанные с обращением клиентов COM к сборкам .NET, а это значит, что в наших примерах клиенты должны к чему-то обращаться. COM-сервером у нас будет притворяться библиотека классов C#. В ней будет опреде-

лен единственный класс — CSharpCal c, в котором будет предусмотрено всего лишь два метода — Add() (для сложения чисел) и Subtract() (для вычитания). Кроме того, этот класс будет реализовывать интерфейс IAdvancedMath, в котором будут предусмотрены методы для выполнения умножения и деления. Само определение класса будет очень простым, однако обратите внимание на применение атрибута Class-Interface:

```
namespace DotNetClassLib
{
        using System;
        using System.Runtime.InteropServices;

        public interface IAdvancedMath
        {
                int Multiply(int x.  int y);
                int  Divide(int x.  int  y);
        }

        [ClassInterface(ClassInterfaceType.AutoDual)]
        public class CSharpCalc : IAdvancedMath
        {
                public CSharpCalc(){}
                public  int  Add(int x.  int y)  {return x + y;}
                public  int  Subtractdnt x.  int  y)  {return x  -  y;}

                int  IAdvancedMath.Multiply(int x.  int y)  {return X*y;}
                int  IAdvancedMath.Divide(int x,  int  y)
                {
                        if(y - - 0)
                                // Исключение будет перехвачено как объект ошибки COM
                                throw new DivideByZeroException();
                        return x / y;
                }
        }
}
```

Генерация библиотеки типов и регистрация типов .NET

Будем считать, что мы откомпилировали проект и в нашем распоряжении появилась готовая сборка. Следующая наша задача — создать для этой сборки промежуточный модуль CCW и зарегистрировать информацию о нем в реестре как о COM-сервере. Это можно сделать двумя способами. Первый способ — воспользоваться утилитой regasm.exe, поставляемой в составе .NET SDK. По умолчанию это средство только заносит данные о регистрации модуля CCW в реестр, однако если указать параметр /tlb, то это средство также создаст необходимую библиотеку типов (то есть модуль CCW):

```
regasm DotNetClassLib.dll /tlb:simpledotnetserver.tlb
```

Другой способ — создать библиотеку типов при помощи утилиты tlbexp.exe, а информацию о созданном модуле занести в реестр уже при помощи утилиты regasm.exe. В любом случае результат будет одним и тем же: в нашем распоряжении появится промежуточный модуль CCW для сборки .NET, и информа-

ция об этом модуле как о COM-сервере будет внесена в реестр операционной системы.

Код приложения DotNetClassLib можно найти в подкаталоге Chapter 12.

Анализ созданной библиотеки типов

Всегда полезно разобраться в том, что для нас было создано автоматически. Давайте загрузим библиотеку типов (файл simpledotnetserver.tlb) в OLE/COM Object Viewer и посмотрим, что у нас получилось. Прежде всего, в Object Viewer можно найти определение IDL для интерфейса класса CSharpCalc (он называется _CSharpCalc):

```
[uuid(AA165958-53F3-3129-83AE-7AE174FE923F), hidden, dual, nonextensible,
            custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
            "DotNetClassLib.CSharpCalc")]
interface _CSharpCalc : IDispatch
{
        // Методы System.Object!
        [id(00000000), propget] BSTR ToString();
        [id(0x60020001)] VARIANT_BOOL Equals([in] VARIANT obj);
        [id(0x60020002)] long GetHashCode();
        [id(0x60020003)] _Type* GetType();

        // Методы интерфейса класса
        [id(0x60020000)] HRESULT Add( [in ] long x. [in] long y. [out, retval] long*
                                                                    pRetVal);
        [id(0x60020001)] HRESULT Subtract( [in ] long x. [in] long y. [out. retval] long*
                                                                    pRetVal);
};
```

В соответствии с заданным нами значением атрибута ClassInterface интерфейс по умолчанию определен как [dual]. (Обратите внимание, что всем членам автоматически присвоены идентификаторы DISPID.) Как мы видим, для интерфейса класса также явно представлены записи для всех членов, унаследованных от System.Object. Более подробно об этом — чуть ниже.

Интересно отметить, что в интерфейсе класса не определены те члены, которые относятся к интерфейсу IAdvancedMath. Причина проста — интерфейс IAdvancedMath реализован явно (подробнее об этом говорилось в главе 4). Однако в сгенерированной библиотеке типов, конечно, содержится и определение IDL для этого пользовательского интерфейса:

```
interface IAdvancedMath : IDispatch
{
        [id(0x60020000)] HRESULT Multiply( [in] long x. [in] long y. [out. retval] long*
                                                                    pRetVal);
        [id(0x60020001)] HRESULT Divide( [in] long x, [in] long y. [out. retval] long*
                                                                    pRetVal);
};
```

Если мы не используем явное наследование интерфейса, то отдельное определение для IadvancedMath тем не менее у нас появится. Однако при этом методы этого интерфейса (Multiply() и Divide()) будут присутствовать и в определении интерфейса класса.

Интерфейс _Object

Сгенерированный код IDL содержит интерфейс с именем _0bject. В этом интерфейсе можно найти все члены System. Object:

```
[uuid(98417C7D-32E8-3FA0-A548-0F0B2EFBE91F), hidden, dual, nonextensible<
                    custom({0F21F359-AB84-41EB-9A78-36D110E6D2F9}, "System.Object")]
dispinterface _Object
{
        properties:
        methods:
        [id(00000000), propget] BSTR ToString();
        [(id(0x60020001)] VARIANT_BOOL Equals([in] VARIANT obj);
        [(id(0x60020002)] long GetHashCode();
        [(id(0x60020003)] _Type* GetType();
};
```

В определение сокласса IDL поддержка этого интерфейса добавляется автоматически:

```
coclass CSharpCalc {
        interface IManageObject;
        [default] interface _CSharpCalc;
        Interface _Object;
        interface IAdvancedMath;
};
```

Вскоре мы увидим применение интерфейса _Object на практике.

Сгенерированное выражение библиотеки

Последнее, что может привлечь наше внимание в сгенерированном модуле CCW, — это выражение библиотеки (library statement). В классическом COM оно используется для представления всех типов, определенных в IDL, которые должны быть помещены в двоичный файл *.tlb. В этом двоичном файле находятся все типы, информация о которых помещена в IDL, и именно этот файл и обеспечивает возможности межъязыкового взаимодействия COM. Правила, согласно которым утилита tlbexp.exe при создании модуля CCW генерирует выражение библиотеки, достаточно просты: выражение библиотеки создается на основе пространств имен .NET.

Мы уже видели, как в сгенерированный двоичный модуль COM переносится информация о типах .NET (интерфейсе класса, IAdvancedMath и т. п.). Интересно отметить, что помимо стандартной информации о типах OLE в выражение библиотеки также помещается информация о mscorlib.dll (главной сборке библиотеки базовых классов ;NET) и mscoree.dll (среды выполнения .NET):

```
[uuid(5C202075-222E-30A4-BF50-4EFC20DDCD27), version(1.0) ]
// На основе названия пространства инен
library DotNetClassLib
{
    '   //TLib: {BED7F4EA-1A96-11D2-8F08-00A0C9A686D}
        importlib("mscorlib.tlb");
        ...
        importlib("mscoree.tlb");
        ...
}
```

Просмотр типов .NET в OLE/COM Object Viewer

В COM предусмотрены категории для компонентов. Категория COM идентифицируется при помощи глобально-уникального идентификатора (GUID), который в этом случае называется CATID (от category identifier). CATID определяет набор взаимосвязанных соклассов. Утилита OLE/COM Object Viewer позволяет группировать классы COM по категориям. Мы это все говорим к тому, что после установки среды выполнения .NET (CLR) в списке установленных на компьютере компонентов COM появляется новый набор компонентов, относящийся к категории .NET (рис. 12.29). В этот набор помещаются те сборки, для которых имеются зарегистрированные промежуточные модули CCW.

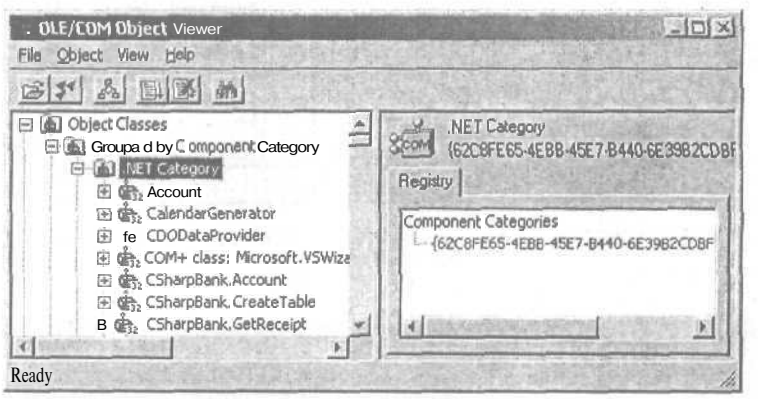


Рис. 12.29. Все сборки .NET, для которых были сгенерированы и зарегистрированы модули CCW, появляются в OLE/COM Object Viewer в категории .NET

Однако если мы попытаемся развернуть узел, относящийся к какой-либо сборке .NET (например, нашей DotNetServerLib), и просмотреть для нее ProgID, то мы получим сообщение об ошибке. Проблема в нашем случае связана с вопросами развертывания — дело в том, что OLE/COM Object Viewer просто не может найти нашу сборку. Вспомним, что для того, чтобы клиент COM смог обратиться к сборке .NET, эта сборка либо должна быть помещена в каталог запуска приложения, либо установлена в глобальный кэш сборок (GAC).

Решить проблему очень просто; достаточно скопировать сборку в один каталог с утилитой OLE/COM Object Viewer или установить сборку в GAC. После этого мы сможем получить о ней полную информацию — например, просмотреть интерфейсы, которые реализованы в ее промежуточном модуле CCW (рис. 12.30).

Просмотр записей в реестре

Перед тем как приступить к созданию клиентов COM, которые будут обращаться к нашей сборке, мы можем также убедиться, что необходимые записи для нее (точнее, для промежуточного модуля CCW) внесены в реестр. Как мы помним, эта

операция выполняется при помощи утилиты regasm.exe. Мы можем провести поиск в реестре по названию сборки и получить ProgID для каждого сокласса, определенного в сборке (рис. 12.31).

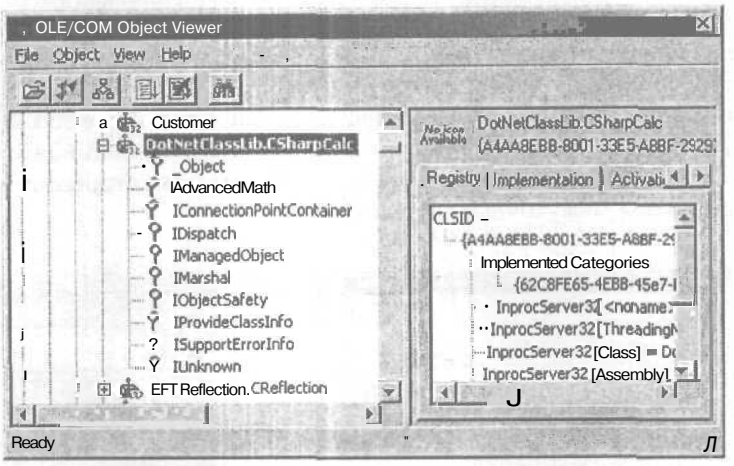


Рис. 12.30. Просмотр интерфейсов, реализованных в промежуточном модуле CCW для нашей сборки

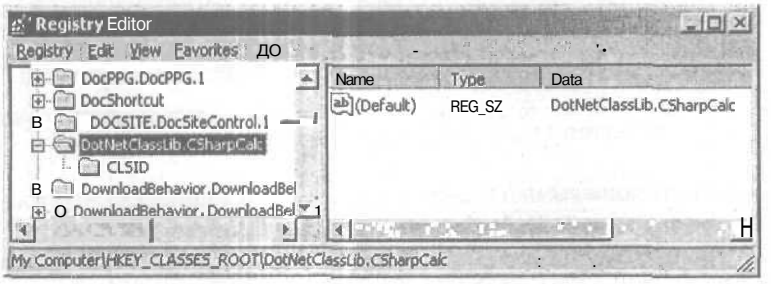


Рис. 12.31. Получаем ProgID для классов нашей сборки

По полученному ProgID мы можем найти, к примеру, значение CLSID (рис. 12.32).

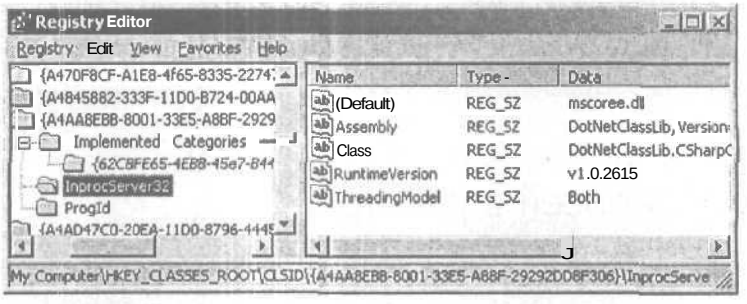


Рис. 12.32. Зарегистрированный CLSID

Когда клиент COM обращается с запросом на активацию COM-сервера к среде выполнения COM (Service Control Manager, SCM), SCM **обращается к** разделу реестра HKCR\CLSID для определения местонахождения зарегистрированного COM-сервера. Б CLSID нашего COM-сервера, как можно видеть на рис. 12.32, существует еще несколько подкаталогов реестра, из которых наиболее важным является подкаталог InprocServer32. В этом подкаталоге должен находиться путь к двоичному файлу COM-сервера, загрузку которого SCM произведет по запросу клиента. Однако для нашей сборки DotNetClassLib.dll мы такого двоичного файла не найдем. Вместо него указан путь к файлу среды выполнения .NET — mscoree.dll (рис. 12.33).

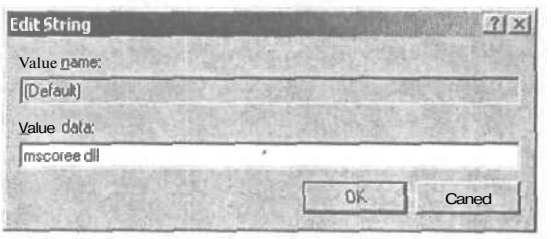


Рис. 12.33. Значение в подкаталоге InprocServer32 указывает на файл среды выполнения .NET

Обратите внимание также на параметр Assembly в том же подкаталоге InprocServer32. Значение этого параметра — полное имя сборки (рис. 12.34).

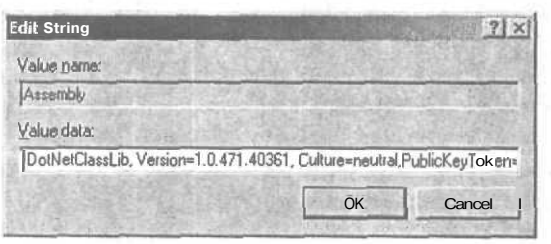


Рис. 12.34. Значение параметра Assembly

Конечно, утилита regasm.exe вносит в реестр и другие записи. Например, поскольку для типов .NET всегда используются интерфейсы, производные от IDispatch, каждый интерфейс настроен на использование oleaut32.dll. Информация о самих типах заносится в раздел реестра HKCR\TypeLib.

Создаем клиента в Visual Basic 6.0

Теперь, когда в нашем распоряжении есть и сборка .NET, и промежуточный модуль CCW к ней, мы должны создать клиентов COM, которые будут обращаться к нашей сборке. Первый наш клиент будет клиентом Visual Basic 6.0. Начнем с того, что создадим новый проект Visual Basic при помощи шаблона Standard EXE и уста-

новим ссылку на сгенерированную библиотеку типов — модуль CCW. Перед тем как начать работать с кодом, сохраним проект в каком-либо месте на диске, чтобы мы могли быстро его найти. После этого скопируем сборку C# 6 тот же самый каталог, в котором будет создаваться клиент Visual Basic (или установим сборку в GAC).

С точки зрения графического интерфейса наш клиент будет предельно простым. Для передачи информации сборке .NET будет использоваться единственная кнопка (объект Button). Результаты вычислений, возвращаемые сборкой .NET, будут выводиться посредством объектов MsgBox. Единственная вещь, о которой не следует забывать: как мы помним, при создании нашей сборки-калькулятора мы предусмотрели генерацию исключения DivideByZeroException на случай, если второй передаваемый параметр метода Divide() интерфейса IAdvancedMath будет равен нулю. Таким образом, наш код должен будет принимать исключение .NET как объект ошибки COM. Сам код клиента приведен ниже (обратите внимание, что мы используем некоторые унаследованные методы, определенные в интерфейсе _Object):

```
Private Sub btnDoEverything_Click()
On Error GoTo OOPS:

    ' Создаем объект .NET и складываем два числа
    ' Dim o As New CSharpCalc
    MsgBox o.Add(30, 30), , "Adding"

    ' Вызываем некоторые методы интерфейса _Object
    MsgBox o.ToString, , "To String"
    MsgBox o.GetHashCode, , "Hash code"

    Dim t As Object
    Set t = o.GetType()
    MsgBox t, , "Type"

    ' Получаем ссылку на пользовательский интерфейс
    ' и заставляем сработать исключение
    Dim i As IAdvancedMath
    Set i = o

    MsgBox i.Multiply(4, 22), , "Multiply"
    MsgBox i.Divide(20, 2), , "Divide"
    MsgBox i.Divide(20, 0)              ' Генерируем ошибку

OOPS:
' Выводим информацию об исключении
MsgBox Err.Description, , "Error!"
End Sub
```

Обратите внимание, что Visual Basic 6.0 не позволяет получить доступ к интерфейсу _Type, возвращаемому при помощи метода _Object.GetType(), поскольку этот интерфейс помечен как скрытый ([hidden]). Вместо этого придется ограничиться гораздо более скромными возможностями System.Object.

Код приложения VBDotNetClient можно найти в подкаталоге Chapter 12.

Некоторые особенности отображения типов .NET в COM

Общая система типов (Common Type System, CTS) допускает применение некоторых возможностей, которые в COM не предусмотрены. Например, классы C# могут поддерживать любое количество конструкторов, перегруженных операторов, перегруженных методов. Кроме того, для классов C# может использоваться классическое наследование. Однако ни одна из этих технологий программирования не может использоваться в COM. Как же производится перевод типов .NET, в которых используются такие возможности, в типы COM при создании промежуточного модуля CCW утилитой ttbexp.exe? Все не так просто.

Давайте рассмотрим такое представление на примере. Предположим, что у нас есть библиотека кода C#, в которой определен один базовый класс и один производный класс. В базовом классе есть переменные, определенные как public (поля), набор конструкторов и единственный виртуальный метод:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class BaseClass
{
        // Переменные
        private int memberVar;
        public string fieldOne;

        // Конструкторы
        public BaseClass(){}
        public BaseClass(int m, string f)
        { memberVal = m; fieldOne = f; }

        // Виртуальный метод
        public virtual void VirMethod()
        { Console.WriteLine("Base VirMethod impl");}
}
```

Производный класс замещает виртуальный метод и определяет еще один метод, который несколько раз перегружен:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class DerivedClass : BaseClass
{
        // Переменная
        public float fieldTwo;

        // Конструктор
        Derived Class(int m. string f) : base(m, f) {}

        // Замещенный метод
        public override void VirMethodO
        {
                Console.WriteLine(""Derived VirMethod impl");
                base.VirMethod();
        }

        // Перегруженные члены
        public void SomeMethod(){}
        public void SomeMethod(int x){}
```

```
        public void SomeMethod(int x, object o){}
        public void SomeMethod(int x, float f){}
}
```

В общем, нам совершенно не важно, что именно делают все эти методы. Гораздо интереснее выяснить, как утилита tLbexp.exe выкрутится из сложной ситуации и создаст для этих конструкций .NET аналоги COM.

Анализ кода COM, сгенерированного для базового класса

Давайте начнем с определения *соклacca* для базового класса .NET (который у нас называется BaseClass). Для этого загрузим сгенерированный файл *.tlb в OLE/COM Object Viewer и найдем там информацию о соклассе:

```
coclass BaseClass
{
        interface IManagedObject;
        [default] interface _BaseClass;
        interface _Object;
};
```

Вряд ли здесь нужно что-то комментировать. Что такое интерфейс класса и для чего нужен интерфейс _0bject — это мы уже обсуждали. Действительно интересные вещи обнаруживаются в определении самого интерфейса класса:

```
Interface _BaseClass : IDispatch
{
        // Методы объекта...

        [id(0x60020004)] HRESULT VirMethod();
        [id(0x60020005), propget] HRESULT fieldOne([out, retval] BSTR* pRetVal);
        [id(0x60020005), propput] HRESULT fieldOne([in] BSTR pRetVal);
};
```

Таким образом, переменные, объявленные как public (то есть поля), представлены в сгенерированной сборке как свойства COM. Это вполне объяснимо, поскольку клиенты COM никогда не получают ссылки на объекты и им всегда приходится работать через ссылки на интерфейсы. А теперь посмотрим, что создано для производного класса.

Анализ кода COM, сгенерированного для производного класса

Как известно, никакого классического наследования между классами в COM не предусмотрено в принципе. Поэтому просто взять и перенести отношение "is-a" между базовым и производным классом не удастся при всем желании. Вместо этого утилита tLbexp.exe делает замечательный фокус: она реализует в соклассе, созданном для производного класса, интерфейс базового класса! Вот так:

```
coclass DerivedClass
{
        Interface IManagedObject;
        [default] interface _DerivedClass;
        Interface _BaseClass;
```

```
        interface _Object:
};
```

Такой подход гарантирует, что в производном классе будут реализованы те же возможности, что и в базовом.

Интерфейс класса для самого класса DerivedClass также представляет интерес. Вспомним, что в исходном определении этого класса в C# была предусмотрена перегрузка одного из методов. Поскольку в COM перегрузка методов в принципе не предусмотрена, то tlbexp.exe приходится использовать радикальные средства:

```
interface _DerivedClass : IDispatch
{
        // Методы интерфейса _Object...

        // Методы, "унаследованные от базового класса"
        [id(0x60020004)] HRESULT VirMethod();
        [id(0x60020005), propget] HRESULT fieldOne([out, retval] BSTR* pRetVal);
        [id(0x60020005), propput] HRESULT fieldOne([in] BSTR pRetVal);

        // "Перегруженный метод"
        [id(0x60020007)] HRESULT SomeMethod();
        [id(0x60020008)] HRESULT SomeMethod_2([in] long x);
        [id(0x60020009)] HRESULT SomeMethod_3([in] long x, [in] VARIANT o);
        [id(0x6002000a)] HRESULT SomeMethod_4([in] long x, [in] single f);

        // Переменная FieldTwo, определенная как public (поле)
        [id(0x6002000b), propget] HRESULT fieldTwo([out, retval] single* pRetVal);
        [id(0x6002000c), propput] HRESULT fieldTwo([in] single pRetVal);
};
```

Итак, провести преобразование таким образом, чтобы все действительно было гладко, не удалось. Вместо одного перегруженного метода SomeMethod() у нас появилось целых четыре, при этом три — с числовыми суффиксами, то есть именами, которые мы им не присваивали. Конечно, это не единственное нарушение, которое может произойти при создании модуля CCW для типа .NET. Подобное безобразие можно будет наблюдать и при использовании вложенных пространств имен, абстрактных базовых классов, структурных типов (перечислений и структур) и т. д. Однако мы не будем углубляться в эти вопросы. Мы познакомились с основными моментами и приемами анализа, а остальную информацию вы сможете получить сами — при помощи электронной документации и путем анализа созданных для ваших типов .NET модулей CCW. Очень рекомендую вам использовать для такого анализа сборку CarLibrary.dll, созданную нами в главе 6... однако мы увлеклись.

Код приложения NetToComIssues можно найти в подкаталоге Chapter 12.

Управление процессом генерации кода IDL (как повлиять на то, что делает утилита tlbexp.exe)

При использовании утилиты tlbimp.exe для генерации прокси-сборки в манифест этой сборки помещается большое количество атрибутов. Если мы создаем сборку

.NET, к которой, как планируется, будут активно обращаться традиционные клиенты COM, мы также можем использовать множество атрибутов (один из них — ClassInterfaceAttribute — мы уже рассматривали). Обычно эти атрибуты используются для того, чтобы управлять процессом создания модуля CCW утилитой tlbimp.exe.

Давайте, как обычно, продемонстрируем применение таких атрибутов на примере. Мы создадим новое пространство имен (AttribDotNetObjects), в котором будет определен единственный интерфейс IBasicMath и единственный класс Calc. Обратите внимание на то, как мы будем использовать в этом пространстве имен атрибуты. При помощи них мы сможем явно определить значение GUID для генерируемых типов и вмешаться в процесс создания аналогов COM для интерфейса IBasicMath и метода Add(). Кроме того, атрибуты будут использованы и для управления процессом представления двух статических функций, определенных в классе Calc. Вот код пространства имен с атрибутами:

```
namespace AttribDotNetObjects
{
    using System;
    using System.Runtime.InteropServices;
    using System.Windows.Forms;

    // Для этого интерфейса .NET мы применяем несколько атрибутов. Они будут
    // использованы утилитой t1bexp.exe при генерации модуля CCW
    [GuidAttribute("47430E06-718D-42C6-9E45-78A99673C43C"),
                    InterfaceTypeAttribute(ComInterfaceType.InterfaceIsDual)]
    public interface IBasicMath
    {
        [DispId(777)] int Add(int x, int y);
    }

    [GuidAttribute("C08F4261-C0C0-46AC-87F3-EDE306984ACC")]
    public class DotNetCalc : IBasicMath
    {
        public DotNetCalc(){}

        public int Add(int x, int y) { return x + y; }

        // Этот атрибут означает, что данный метод должен быть вызван
        // при регистрации модуля CCW
        [ComRegisterFunctionAttribute]
        public static void AddExtraRegLogic(string regLoc)
        {
            // Выполняем дополнительные действия при регистрации
            MessageBox.Show("Inside AddExtraLogic f(x)", ".NET assembly
                                                                says:");
        }

        // Этот атрибут означает, что данный метод будет вызван при удалении
        // регистрации модуля CCW
        [ComUnregisterFunctionAttribute]
        public static void RemoveExtraRegLogic(string regLoc)
        {
            // Выполняем дополнительные действия при удалении регистрации
```

```
                MessageBox.Show(" Inside RemoveExtraRegLogic f(x)", ".NET assembly
                                                                says:");
            }
        }
    }
```

Код приложения AttribDotNetObjects можно найти в подкаталоге Chapter 12.

Что у нас получилось

Если мы создадим для нашей сборки промежуточный модуль CCW при помощи утилиты tlbexp.exe, то мы сможем обнаружить, что наши атрибуты подействовали. Для IID и CLSID были использованы явно указанные нами значения, а интерфейс IBasicMath определен как [dual]. Вот код IDL для интерфейса IBasicMath:

```
// В сборке мы использовали атрибуты:
   [GuidAttribute("47430E06-718D-42C6-9E45-78A99673C43C"),
   InterfaceTypeAttribute(ComInterfaceType.InterfaceIsDual)]


// Полученный код IDL:
[odl, uuid(47430E06-718D-42C6-9E45-78A99673C43C), dual, oleautomation, custom({0F21F359-
                    AB84-41E8-9A78-36D110E6D2F9}, "AttribDotNetObjects.IBasicMath")]
interface IBasicMath : IDispatch
{
        [id(0x00000309)]            // У нас было: [DispId(777)]
        HRESULT AddC [in] long x, [in] long y, [out, retval] long* pRetVal);
};
```

Если мы вернемся к представлению интерфейса IBasicMath в коде C# и изменим значение атрибута InterfaceTypeAttribute следующим образом:

```
[GuidAttribute("47430E06-718D-42C6-9E45-78A99673C43C"),
InterfaceTypeAttribute(ComInterfaceType.InterfaceIsUnknown)]
        public interface IBasicMath
{
        int Add(int x, int y);
}
```

то код IDL также изменится:

```
// Интерфейс теперь не [dial]!
[
        odl, uuid(47430E06-718D-42C6-9E45-78A99673C43C), oleautomation, custom({0F21F359-
                                                    AB84-41E8-9A78-36D110E6D2F9},
                                                    "AttribDotNetObjects.IBasicMath")
]
interface IBasicMath : IUnknown
{
        // Следов DispID не осталось
        HRESULT _stdcall Add([in] long x, [in] long y, [out, retval] long* pRetVal);
};
```

Управление регистрацией промежуточного модуля CCW

Возможно, об атрибутах ComRegisterFunctionAttribute и ComUnregisterFunctionAttribute следует рассказать подробнее. Как вы, наверное, уже знаете, в классических COM-

серверах определены две функции - DllRegisterServer и DllUnregisterServer. Эти функции вызываются самыми разными утилитами, осуществляющими регистрацию сервера COM в реестре и удаление такой регистрации. В сборках .NET таких функций нет.

Если нам потребуется производить регистрацию промежуточного модуля CCW, используемого для доступа к нашей сборке, каким-либо специальным образом, мы можем добавить в эти функции в генерируемом COM-сервере свои действия. Для этого нам потребуется определить статические методы с вышеуказанными атрибутами. Например, при регистрации промежуточного модуля для сборки из предыдущего примера мы получим сообщение, представленное на рис. 12.35.

Рис. 12.35. Управление регистрацией COM

Необходимо отметить два момента, связанных с применением этих атрибутов. Во-первых, названия методов, помеченных этими атрибутами, могут быть какими угодно, но метод, помеченный атрибутом ComRegisterFunctionAttribute, должен принимать параметр типа String. Во-вторых, если мы определили один метод и пометили его атрибутом ComRegisterFunctionAttribute, то мы должны определить и второй метод для ComUnregisterFunctionAttribute.

Взаимодействие со службами COM+

Последняя часть этой главы будет посвящена тому, как создавать типы .NET, использующие возможности среды выполнения COM+. Однако прежде чем приступить к изложению конкретных особенностей применения библиотеки базовых классов для этой цели, мне придется вначале дать общий обзор служб COM+ и предыстории их появления.

Наверное, вам знаком продукт, который называется Microsoft Transaction Server (MTS). MTS — это сервер приложений, на котором могут размещаться модули классического COM для целей построения многоуровневых приложений уровня предприятия. Например, предположим, что мы создали классический COM-сервер, который должен обращаться к источнику данных (например, через ADO) и вносить изменения в несколько взаимосвязанных таблиц. Если этот COM-сервер будет установлен под MTS, он получит важные дополнительные возможности, например поддержку использования явно объявленных транзакций или весьма удобной в применении модели безопасности, основанной на ролях.

Для каждого COM-класса, предназначенного для работы под MTS, используется специальный контекстный объект, который определяет, как именно будет применяться объект MTS. Например, такой контекстный объект может хранить информацию о правах пользователя в системе безопасности, об участии данного

объекта в транзакции, о том, что COM-класс больше не нужен и его можно удалять из оперативной памяти и т. д.

В большинстве случаев типы COM, которые предназначаются для работы под управлением сервера MTS, являются сущностями без состояния. Это значит, что объект создается и удаляется средой выполнения MTS (для более эффективного управления ресурсами операционной системы) и при этом подключенный клиент не затрагивается. Если клиент производит вызов объекта COM-сервера, ссылкой на который, как ему кажется, он обладает, сервер MTS просто создает для этого клиента новый объект COM-сервера. Для COM-серверов под MTS обычно не предусматривается никакого графического интерфейса: они выполняют роль традиционных бизнес-объектов, которые занимаются лишь обслуживанием запросов со стороны клиентов.

MTS — это замечательное средство для создания очень надежных и масштабируемых систем, однако у него есть недостатки. Например, интеграцию между средами выполнения COM и MTS нельзя назвать идеальной. Каждая среда выполнения вносит записи о своих модулях в свою часть реестра, и в результате модуль COM, установленный под MTS, не будет работать как обычный in-proc сервер. Кроме того, механизм создания объектов в классическом COM отличается от механизма создания объектов в MTS. Модули COM, установленные под MTS, должны создавать другие объекты специально для работы под MTS, используя для этого весьма специфические методы.

COM+ — это технология программирования, которая объединила возможности MTS и классического COM в единое целое, добавив многие важные возможности и устранив недостатки этих двух систем. Приложения COM+ могут использовать все возможности, которые имелись в MTS (явно объявленные транзакции, механизм безопасности, основанный на ролях и т. п.), а также многие новые. Вот краткий перечень основных достоинств COM+:

- COM+ поддерживает пулы объектов. Среда выполнения COM+ может хранить в памяти наборы объектов, которые быстро предоставляются клиенту по его запросу. Это позволяет гораздо быстрее возвращать пользователю ссылки на интерфейсы для объектов COM+. Однако нерациональное использование этого средства может привести к перерасходу оперативной памяти на компьютере, где работает сервер COM+.
- В COM+ реализована новая модель событий, которая называется LCE (Loosely Coupled Events, слабосвязанные события). Эта модель позволяет пользователям и клиентам взаимодействовать, не будучи постоянно связанными друг с другом. Класс COM+ при использовании этой модели событий может отправлять сообщение о событии, не заботясь о том, подключен ли кто-либо к нему напрямую или нет. В то же время клиент COM+ может получать сообщения о событиях, не будучи постоянно подключен к серверу.
- В COM+ предусмотрены так называемые строки создания объектов (object construction strings). Как мы помним, в классическом COM клиент не мог вызывать конструктор COM-сервера напрямую. В COM+ появился новый интерфейс IObjectConstruct, который позволяет передавать параметры, используемые при создании объекта в форме значения BSTR.

- В COM+ появилась возможность работы с очередями сообщений на уровне объявлений. Раньше для работы с очередями приходилось организовывать довольно хитрое взаимодействие с Microsoft Message Queue (MSMQ). Теперь в состав COM+ входят так называемые Queued Components (QC), с которыми работать гораздо проще.

Вывод, к которому я хотел вас подвести, таков: применение COM+ позволяет резко упростить создание распределенных приложений. Единственная проблема Заключается в том, что службы COM+ изначально предназначались для использования традиционными двоичными модулями COM. Чтобы возможности COM+ стали доступны для типов .NET, нам потребуется воспользоваться специально разработанными для этой цели типами из пространства имен System.EnterpriseServices.

Пространство имен System.EnterpriseServices

Для создания типов .NET, которые смогут использоваться в среде выполнения COM+, нам потребуется поместить в тип .NET значительное количество атрибутов из пространства имен System.EnterpriseServi ces. Если у вас есть опыт работы с классическим MTS и (или) COM+, то эти атрибуты покажутся вам очень знакомыми. Наиболее важные из них представлены в табл. 12.11.

Таблица 12.11. Наиболее важные типы пространства имен System.EnterpriseServices

Тип	Описание
ApplicationActivationAttribute	Определяет, будет ли компонент, определенный в сборке, запускаться в процессе создателя (библиотечное приложение) или в системном процессе (серверное приложение)
ApplicationIDAttribute	Определяет идентификатор приложения сборки (аналогично GUID)
ApplicationQueingAttribute InterfaceQueingAttribute	Используются для реализации поддержки QC (Queued Components)
AutoCompleteAttribute	Помечает метод как AutoComplete. Если метод завершает свою работу нормально, вызывается метод SetComplete(). Если в процессе его выполнения возникло исключение, то автоматически вызывается метод SetAbort()
ComponentAccessControlAttribute	Включает проверки системы безопасности для вызовов заданного объекта
ConstructionEnabledAttribute	Включает поддержку средств конструирования объектов COM+
ContextUtil	Рекомендуемый метод для получения информации о контексте объекта COM+ версии 1.0. Этот тип определяет несколько статических членов, которые позволяют получать информацию о контексте объекта
DescriptionAttribute	Позволяет присвоить описание оборке (приложению), компоненту, методу или интерфейсу
EventClassAttribute EventTrackingEnabledAttribute	Используются для организации взаимодействия с LCE (Loosely Coupled Events)
JustInTimeActivationAttribute	Включает (или отключает) активацию JIT — Just-In-Time

Тип	Описание
SecurityCallContext SecurityCallers SecurityIdentifier SecurityIdentity SecurityRoleAttribute	Используются для организации взаимодействия типов .NET с моделью безопасности MTS/COM+, основанной на ролях
SharedPropertyGroupManager SharedPropertyGroup SharedProperty	Обеспечивают доступ к Shared Property Manager (SPM)
TransactionAttribute	Определяет типы транзакций, которые могут быть использованы для объекта. Должны использоваться значения из перечисления TransactionOption

Особенности создания типов .NET для работы под COM+

Можно сказать, что для создания сборки .NET, которая будет нормально работать под COM+, нам потребуется следовать целому сборнику рецептов.

Самое главное: каждый класс .NET, предназначенный для работы под COM+, должен происходить от базового класса System.ServicedComponent. Этот базовый класс обеспечивает реализацию по умолчанию для классического интерфейса MTS IObjectControl (с методами Activate(), Deactivate() и CanBePooled()). Если мы решили, что нам нужны собственные реализации этих методов — пожалуйста, можно их замещать.

Конечно же, нам потребуется поместить в наши типы .NET значительное количество атрибутов для уточнения особенностей поведения сборки под COM+. После этого мы должны откомпилировать сборку и использовать утилиту regsvcs.exe. С работой этой утилиты мы еще познакомимся, а пока скажем лишь, что она выполняет множество действий, необходимых для установки наших типов в каталог COM+.

Кроме того, мы должны поместить созданную нами сборку .NET в GAC (как это делается, было рассказано в главе 6). Объяснение очень простое: файлу dllhost.exe (это исполняемый модуль COM+) необходимо иметь возможность находить нашу сборку. Согласитесь, что наиболее логичный способ это обеспечить — установить сборку в GAC.

Теперь, когда с теорией мы познакомились, настало время создать сборку .NET, которая будет обслуживать клиентов COM+.

Пример класса C# для работы под COM+

Для наших целей мы создадим новую библиотеку кода, которая будет называться DotNetCOMPlusServer (в среде выполнения .NET могут работать только модули DLL). В этой библиотеке кода будет определен единственный класс — ComPlusType со следующими характеристиками:

- этот класс будетподдерживать строку конструирования (constructorstring);
- объекты этого класса можно будет объединять в пул. Верхний предел пула мы установим равным 100, а нижний — 5;

- в классе будет определен единственный метод, который может выполниться успешно, а может вернуть сообщение об ошибке. Чтобы получать сведения о результате выполнения, мы используем атрибут AutoComplete.

Вот код C# для этого класса:

```
// Добавим ссылку на System.EnterpriseServices.dll!
using System.EnterpriseServices;
using System.Windows.Forms;
...

// Объекты этого класса можно Судет объединять в пул
[ObjectPooling(true, 5, 100)]
// Этот класс поддерживает строку конструирования
[ConstructionEnabledAttribute(true)]
[ClassInterface(ClassInterFaceType.AutoDual)]
public class ComPlusType : ServicedComponent, IObjectConstruct
{
        // Реализуем интерфейс IObjectConstruct
        public void Construct(object o)
        {
                // Получаем ссылку на интерфейс IObjectConstructionString
                IObjectConstructionString ics = (IObjectConstructionString)o;
                MessageBox.Show(ics.ConstructString, "Ctor string is");
        }

        // Реализуем унаследованные абстрактные члены
        public override void Activate()
        { MessageBox.Show("In activate!"); }

        public override void Deactivate()
        { MessageBox.ShowC'In deactivate!"); }

        public override bool CanBePooled()
        { return true; }

        public ComPlusType(){}

        // Единственный метод нашего класса
        public void DeleteCar(int id)
        {
                MessageBox.Show("Deleting car number " + id.ToString(), "Delete Car");
        }
}
```

Кончено, наш класс не назовешь очень сложным, и он не выполняет никаких действий, которые были бы положены классам, работающим под COM+ (например, он не удаляет автомобили из хранилища данных). Однако для наших целей такого класса вполне достаточно.

Поскольку мы установим нашу сборку в GAC, нам потребуется сделать еще одно дело — присвоить ей «сильное имя». Как уже говорилось в главе 6, для этой цели используется утилита sn.exe. Создадим файл *.snk и в файл AssemblyInfo.cs добавим следующий атрибут уровня сборки (только не забудем указать путь к файлу *.snk на нашем компьютере):

```
[assembly: AssemblyKeyFile(@"D:\DotNetComPlusServer\bin\Debug\thekey.snk")]
```

Кроме того, мы можем явно указать номер версии и редакции нашей сборки (чтобы они не изменялись):

```
[assembly : AssemblyVersion("1.0.0.0")]
```

Теперь сборку можно компилировать.

Добавление атрибутов уровня сборки для COM+

Наша сборка вполне созрела для того, чтобы добавить ее в каталог COM+. Утилита regsvcs.exe автоматически умеет генерировать AppID и имя приложения. Однако если нам необходимо явно определить эти и другие параметры, мы можем поместить в файл AssemblyInfo.cs следующие атрибуты уровня сборки:

```
[ assembly:ApplicationActivation(ActivationOption.Server)]
[ assembly: ApplicationID("4fb2d46f-efc8-4643-bcd0-6e5bfa6a174c")]
C assembly: ApplicationName("DotNetComPlusServer")]
[ assembly: Description("This app really kicks.")]
```

Атрибут ApplicationID, конечно, определяет GUID создаваемого нами приложения COM+. Вряд ли нужно каким-то образом комментировать атрибуты ApplicationName и Description. Единственное, что нужно объяснить, — для чего нужен атрибут ApplicationActivation. Как уже говорилось, приложения MTS и COM+ бывают двух разновидностей: библиотечные (активируемые в процессе вызывающего клиента) и серверные (для них создается новый объект dllhost.exe). По умолчанию создается библиотечное приложение. Если мы хотим использовать другой тип, то мы, как в нашем примере, можем указать для этого атрибута значение ActivationOption.Server.

Помещаем сборку в каталог COM+

Чтобы поместить сборку .NET в каталог COM+, нам потребуется вначале сгенерировать для нее промежуточный модуль CCW (библиотеку типов) при помощи tlbexp.exe, а затем зарегистрировать этот тип в системном реестре при помощи regasm.exe. Но в отличие от сборки .NET, притворяющейся обычным COM-сервером, для COM+ этого недостаточно. Нам нужно еще внести необходимую информацию в каталог COM+ (RegDB).

Однако всего этого можно и не делать. В составе .NET SDK поставляется утилита regsvcs.exe, которая выполняет все эти операции автоматически. Если быть точным, то она выполняет следующие действия:

- загружает сборку в оперативную память;
- заносит необходимую информацию в реестр операционной системы (вместо regasm.exe);
- генерирует и регистрирует библиотеку типов (вместо tlbexp.exe);
- устанавливает сгенерированную библиотеку типов в каталог COM+;
- делает все это в соответствии с указаниями, которые мы поместили в код сборки .NET в виде атрибутов.

Для утилиты regsvcs.exe предусмотрено множество параметров командной строки, но простейший вариант ее применения таков:

```
regsvcs /fc DotNetComPlusServer.dll
```

Параметр /fc означает «find or create» (найди или создай) — то есть если такое приложение COM+ не обнаружится в каталоге COM+, оно будет туда помещено. Мы можем также указать в качестве параметра командной строки имя генерируемой промежуточной библиотеки типов. Если оно определено не будет (как в нашем случае), имя генерируемой библиотеке будет присвоено автоматически на основе имени сборки .NET. Последнее, что нам осталось сделать, — поместить сборку .NET в GAC, если мы еще не сделали этого раньше (рис. 12.36).

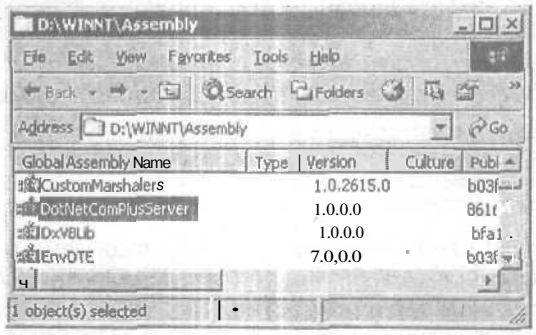


Рис. 12.36. Устанавливаем сборку .NET для работы под COM+ в GAC

Проверяем установку при помощи Component Services Explorer

После того как все операции выполнены, мы можем убедиться в том, что приложение действительно помещено в каталог COM+. Для этого можно использовать утилиту, которая называется Component Services Explorer (рис. 12.37).

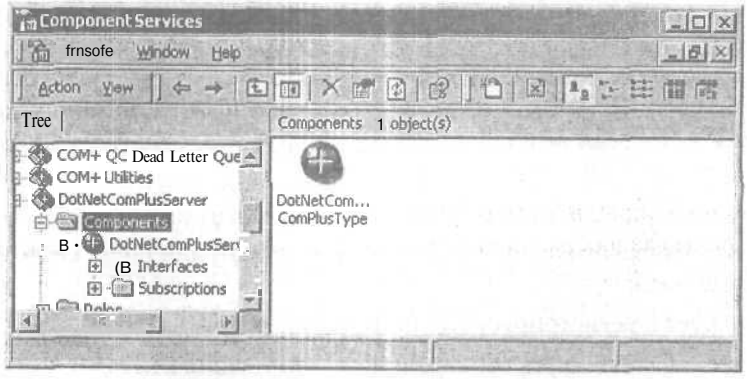


Рис. 12.37. Наше приложение .NET, имеющее значок, напоминающий таблетку аспирина, в окне Component Services Explorer

Если открыть свойства нашего приложения, то можно убедиться в том, что наши атрибуты были восприняты как положено. Например, откроем вкладку Activation (Активация) свойств нашего приложения (рис. 12.38).

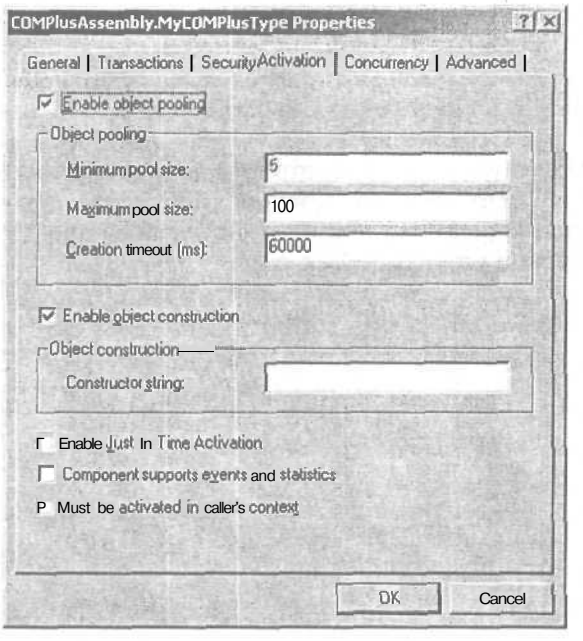


Рис. 12.38. Отображение свойств приложения на вкладке Activation

Значения, которые показаны на этой вкладке, были сгенерированы на основе следующего кода нашего класса C#:

```
// Объекты этого класса можно будет объединять в пул
[ObjectPooling(true, 5, 100)]
// Этот класс поддерживает строку конструирования
[ConstructionEnabledAttribute(true)]
public class ComPlusType : ServicedComponent, IObjectConstruct
{...}
```

Код приложения DotNetComPlusServer можно найти в подкаталоге Chapter 12,

Подведение итогов

.NET — это замечательная вещь. Однако очень часто приходится совместно использовать и сборки .NET, и модули с традиционным двоичным кодом. В .NET предусмотрены разнообразные и мощные средства, которые позволяют коду .NET успешно взаимодействовать с традиционным кодом (и наоборот).

В самом начале этой главы были рассмотрены службы PInvoke — наиболее примитивное средство из богатого набора .NET. Службы PInvoke позволяют производить вызовы к API Win32 и пользовательским модулям DLL.

Большая часть этой главы была посвящена описанию того, как типы .NET могут обращаться к традиционным COM-серверам. Как мы выяснили, для этого потребуется сгенерировать для интересующего нас COM-сервера промежуточную прокси-сборку RCW. Б прокси-сборке предусмотрены аналоги членов COM-сервера. При обращении к этим аналогам вызовы клиента .NET передаются исходному COM-серверу.

Далее в этой главе были рассмотрены средства для решения обратной задачи — как традиционные клиенты COM могут обратиться к сборке .NET. Проблема решается очень похожими средствами — необходимо создать промежуточный модуль CCW, который будет перенаправлять запросы традиционных клиентов COM сборке .NET.

В самом конце этой главы мы рассмотрели еще один важный вопрос — как при помощи средств .NET создавать приложения, работающие под управлением среды COM+. Эта задача решается при помощи типов из пространства имен System.EnterpriseServices и утилиты regsvcs.exe.

Доступ к данным при помощи ADO.NET

13

Если только вы не занимаетесь исключительно созданием «стрелялок», вам, скорее всего, потребуется обращаться к базам данных. Как, наверное, вы уже догадываетесь, в платформе .NET определено множество типов (организованных в соответствующие пространства имен), которые помогут нам обеспечить взаимодействие с локальными и удаленными хранилищами данных. Общее название пространств имен с этими типами — ADO.NET. Как мы увидим, это — не просто перенос классической модели ADO на платформу .NET, но ее коренная переработка.

В начале этой главы будут рассмотрены некоторые наиболее важные типы из пространства имен System.Data — такие как DataColumn, DataRow и DataTable. Эти классы обеспечивают возможность взаимодействия с наборами данных, помещенных в память локального компьютера. После этого мы подробно ознакомимся с главным типом ADO.NET — DataSet. DataSet — это помещаемое в память компьютера представление о наборе взаимосвязанных таблиц. Мы узнаем, каким образом можно программно моделировать отношения между таблицами, создавать пользовательские представления и производить запросы. После этого мы узнаем, как можно создавать объекты Dataset, заполненные данными из распространенных СУБД, таких как MS SQL Server, Oracle и MS Access. В самом конце главы мы рассмотрим концепцию управляемых поставщиков (managed providers) .NET и типы OleDbDataAdapter и SqlDataAdapter.

Почему потребовалось создавать ADO.NET

Первое, что необходимо сказать про ADO.NET, — то, что это не просто улучшенная и расширенная версия классического ADO. У традиционного ADO и ADO.NET существуют как схожие черты (концепция объектов «соединения» и «командных» объектов), так и существенные различия (к примеру, в ADO.NET вы уже не найдете такого важного типа, как Recordset). Самые важные типы ADO.NET, в свою очередь, не имеют эквивалентов в мире классического ADO (это справедливо, например, для DataSet).

ADO.NET — это новая технология доступа к базам данных, специально оптимизированная для нужд построения рассоединенных (disconnected) систем на платформе .NET. Разработчики ADO.NET ориентировались на приложения N-tier — архитектуру многоуровневых приложений, которая в настоящее время стала фактически стандартом для создания распределенных систем.

Подчеркнем отличия классического ADO от ADO.NET: «старое» ADO было ориентировано прежде всего на создание клиент-серверных приложений, когда клиент и сервер должны постоянно взаимодействовать друг с другом. ADO.NET расширяет концепцию *объектов-наборов* записей в базе данных новым типом DataSet, который представляет локальную копию сразу множества взаимосвязанных таблиц. При помощи объекта DataSet пользователь может локально производить различные операции с содержимым базы данных, будучи физически рассоединен с СУБД, и после завершения этих операций передавать внесенные изменения в базу данных при помощи соответствующего «адаптера данных» (data adapter).

Еще одно существенное различие между «просто ADO» и ADO.NET заключается в том, что в ADO.NET реализована полная поддержка представления данных в XML-совместимых форматах. Если заглянуть «внутрь» ADO.NET, то выяснится, что закачанные для локальной обработки наборы данных представлены именно как XML (в этом же формате они и передаются с сервера баз данных). Поскольку данные в форматах XML очень удобно передавать при помощи обычного HTTP, сразу же решаются многие проблемы с установлением соединений через брандмауэры.

В классическом ADO для перемещения данных между уровнями использовался протокол маршалинга COM. Этот протокол имеет множество неоспоримых преимуществ, но у него есть и серьезные ограничения. Например, большинство брандмауэров не будут пропускать пакеты RPC при установлении соединений по этому протоколу, в результате чего развертывание многоуровневого приложения в реальных условиях предприятия может оказаться очень непростой задачей.

Еще одно важнейшее различие между классическим ADO и ADO.NET заключается в том, что ADO.NET — это библиотека управляемого кода и взаимодействие с ней производится как с обычной сборкой .NET. Типы ADO.NET используют возможности управления памятью CLR и могут использоваться во многих .NET-совместимых языках. При этом обращение к типам ADO.NET (и их членам) производится практически одинаково вне зависимости от того, какой язык используется.

ADO.NET: общая картина

Все типы ADO.NET предназначены для выполнения единого набора задач: установить соединение с хранилищем данных, создать и заполнить данными объект DataSet, отключиться от хранилища дачных и вернуть изменения, внесенные в объект DataSet, обратно в хранилище данных. Объект DataSet — это очень важный и интересный тип данных, представляющий локальный набор таблиц и информацию об отношениях между ними. В некоторых отношениях DataSet очень напоминает рассоединенный Recordset из «старого» ADO. Главное различие между рассоединенным Recordset и DataSet заключается в том, что Recordset представляет собой единственную таблицу, а DataSet — набор связанных таблиц. На практике нам ничто не мешает создать на клиенте объект DataSet, который будет представлять *полную копию* удаленной базы данных.

После создания объекта DataSet и его заполнения данными мы можем программными средствами производить запросы к нему и перемещаться по таблицам. Мы можем выполнять все операции, как при работе с обычными базами данных: добавлять в таблицы новые записи, удалять и изменять существующие, применять к ним фильтры и т. п. После того как клиент завершит внесение изменений, информация о них будет отправлена в хранилище данных для обработки.

Скорее всего, одним из первых вопросов у каждого, кто начал разбираться с ADO.NET, будет: «А как создать DataSet?» Ответ будет звучать так; при помощи управляемого провайдера (managed provider). Управляемый провайдер — это набор классов, реализующих интерфейсы, определенные в пространстве имен System.Data. Речь идет об интерфейсах IDbCommand, IDbDataAdapter, IDbConnection и IDataReader (рис. 13.1).

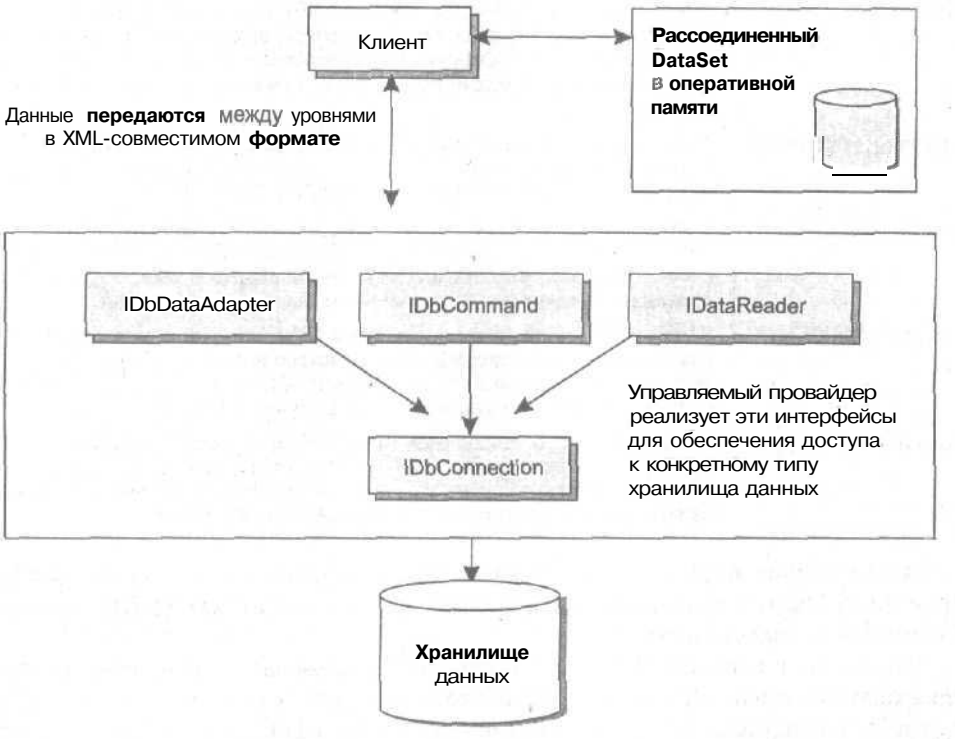


Рис. 13.1. Взаимодействие клиента с управляемыми провайдерами

В состав ADO.NET включены два управляемых провайдера: провайдер SQL и провайдер OleDb. Провайдер SQL специально оптимизирован под взаимодействие с Microsoft SQL Server версии 7.0 и последующих. Для других источников данных предлагается использовать провайдер OleDb, который можно использовать для обращения к любым хранилищам данных, поддерживающим протокол OLE DB. Однако учтите, что провайдер OleDb работает при помощи «родного» OLE DB и требует возможности взаимодействия при помощи COM.

Конечно же, в самом скором будущем появятся управляемые провайдеры от производителей СУБД, которые будут эффективно взаимодействовать со «свои-

ми» источниками данных. Однако до этого времени в большинстве случаев придется использовать OleDb провайдер.

Знакомство с пространствами имен ADO.NET

Все возможности ADO.NET заключены в типах, определенных в соответствующих пространствах имен. Краткий обзор главных пространств имен ADO.NET представлен в табл. 13.1.

Таблица 13.1. Пространства имен ADO.NET

Пространство имен	Описание
System.Data	Это — главное пространство имен ADO.NET. в нем определены типы, представляющие таблицы, столбцы, записи, ограничения и, конечно же, самый важный тип — DataSet. В этом пространстве имен нет типов для подключения к источнику данных — только типы, представляющие сами данные
System.Data.Common	Здесь определены типы, общие для всех управляемых провайдеров, Многие из них выступают в качестве базовых классов для классов из пространств имен для провайдеров SQL и OleDb
System.Data.OleDb	В этом пространстве имен определены типы для установления соединений с OLE DB-совместимыми источниками данных, выполнения к ним SQL-запросов и заполнения данными объектов DataSet. Типы этого пространства имен очень похожи на типы классического ADO
System.Data.SqlClient	В этом пространстве имен определены типы, которые и составляют управляемый провайдер SQL С их помощью можно выполнять очень эффективное взаимодействие с Microsoft SQL Server напрямую, минуя промежуточные преобразования (как в случае использования OleDb)
System.Data.SqlTypes	Представляют собой «родные» типы данных Microsoft SQL Server. Конечно же, можно использовать и стандартные типы данных CLR, но использование типов из этих пространств имен позволяет добиться наивысшей производительности при работе с SQL Server

Все пространства имен ADO.NET расположены в одной сборке — System.Data.dll (рис. 13.2). Это означает, что в любом проекте, использующем ADO.NET, мы должны добавить ссылку на эту сборку.

В любом приложении ADO.NET мы должны использовать по крайней мере одно пространство имен — System. Data. Кроме того, нам практически во всех ситуациях потребуется использовать еще либо пространство имен System.Data.OleDb или System.Data.SqlClient — для установления соединения с источником данных. Подробнее о необходимых условиях для создания приложения ADO.NET мы поговорим ниже, а сейчас настала очередь подробно рассмотреть типы основных пространств имен.

Типы пространства имен System.Data

Эти типы предназначены для представления данных, полученных из источника (но не для установления соединения непосредственно с источником). В основном эти типы представляют собой объектные представления примитивов для работы с базами данных — таблицами, строками, столбцами, ограничениями и т. п. Наи-

более часто используемые типы System.Data представлены в табл. 13.2. Кроме того, в этом пространстве имен определены важные исключения, которые могут быть сгенерированы при работе с базами данных (NoNullAllowedException, RowNotInTableException, MissingPrimaryKeyException и т. п.).

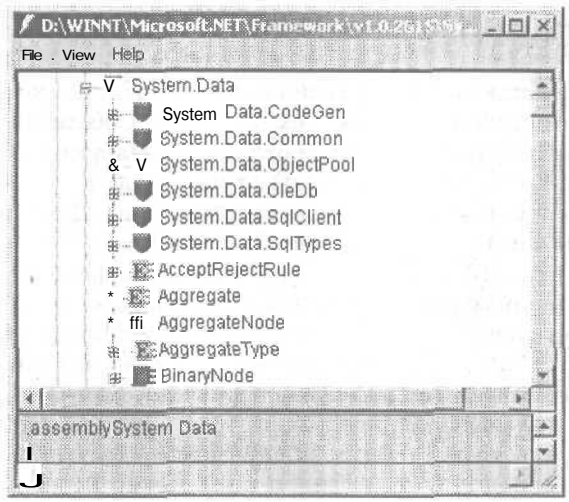


Рис. 13.2. Сборка System.Data.dll

Таблица 13.2. Типы пространства имен System.Data

Тип	Назначение
DataColumnCollection DataColumn	DataColumn представляет один столбец в объекте DataTable, DataColumnCollection — все столбцы
ConstraintCollection Constraint	Constraint — объектно-ориентированная оболочка вокруг ограничения (например, внешнего ключа или уникальности), наложенного на один или несколько DataColumn, ConstraintCollection — все ограничения в объекте DataTable
DataRowCollection DataRow	DataRow представляет единственную строку в DataTable, DataRowCollection — все строки в DataTable
DataRowView DataView	DataRowView позволяет создавать настроенное представление единственной строки, DataView — созданное программным образом представление объекта DataTable, которое может быть использовано для сортировки, фильтрации, поиска, редактирования и перемещения
DataSet	Объект, создаваемый в оперативной памяти на клиентском компьютере. DataSet состоит из множества объектов DataTable и информации об отношениях между ними
ForeignKeyConstraint UniqueConstraint	ForeignKeyConstraint представляет ограничение, налагаемое на набор столбцов в таблицах, связанных отношениями первичный — внешний ключ. UniqueConstraint — ограничение, при помощи которого гарантируется, что в столбце не будет повторяющихся записей
DataRelationCollection DataRelation	Тип DataRelationCollection представляет набор всех отношений (то есть объектов DataRelation) между таблицами в DataSet
DataTableCollection DataTable	Тип DataTableCollection представляет набор всех таблиц (объектов DataTable) в DataSet

В соответствии с возникшей в предыдущих главах традицией, вначале мы посмотрим, как работать с типами из System.Data вручную. После того как мы научимся создавать населенный объект DataSet вручную, нам не составит труда разобраться, как именно работает с объектами DataSet управляемый провайдер.

Тип DataColumn

Тип DataColumn представляет отдельный столбец в таблице (которая, в свою очередь, должна быть представлена объектом DataTable). Собственно говоря, столбцы с необходимыми атрибутами и составляют таблицу. Например, предположим, что у нас есть таблица Employees (сотрудники) с тремя столбцами: EmpID (идентификатор сотрудника), FirstName (имя) и LastName (фамилия). Для представления данных этой таблицы нам потребуются три объекта DataColumn — по числу столбцов. Как мы вскоре увидим, объект DataTable работает с внутренней коллекцией типов DataColumn, доступ к которой производится через свойство Column.

Если у вас есть подготовка в области теории баз данных, вам, без сомнения, известно, что на столбцы в таблице могут быть наложены ограничения. Например, столбец может быть определен в качестве первичного ключа, ему может быть назначено значение по умолчанию, его можно сделать доступным только для чтения и т. п. Кроме того, каждому столбцу в таблице должен соответствовать определенный тип данных (например, int, varchar и т. п.). В нашей таблице Employees мы, например, можем использовать для EmpID целочисленный тип данных int, а для FirstName и LastName — символьные массивы varchar. Для того чтобы реализовать возможности, связанные с ограничениями, в классе DataColumn предусмотрено большое число свойств. Наиболее важные из них представлены в табл. 13.3.

Таблица 13.3. Свойства класса DataColumn

Свойство	Описание
AllowDbNull	Используется для определения того, может ли столбец содержать значения типа NULL (пустые значения). По умолчанию — может (значение этого свойства равно true)
AutoIncrement AutoIncrementSeed AutoIncrementStep	Эти свойства используются для настройки автоматического приращения значений в столбце. Это может быть полезно, если необходимо обеспечить уникальность значений в столбце (например, для первичного ключа). По умолчанию автоматическое приращение значений в столбцах отключено
Caption	Определяет заголовок столбца для отображения в пользовательском приложении (например, этот заголовок может быть использован в DataGrid)
ColumnMapping	Определяет, как будет представлен столбец (объект DataColumn) при сохранении DataSet в формате XML (при использовании метода DataSet.WriteXml())
ColumnName	Позволяет получить или установить имя столбца в коллекции Columns (внутренняя коллекция для столбцов в DataTable). Если имя столбца не определено явно, будут использованы значения по умолчанию: Column1, Column2, Column3 и т. д.
DataType	Определяет тип данных (boolean, string, float и т. п.), используемый для значений в столбце
DefaultValue	Позволяет установить или получить значение по умолчанию для столбца. Это значение будет автоматически использовано, если при вставке новой строки не укажете явно другое значение

Свойство	Описание
Expression	Позволяет получить или установить выражение, используемое для фильтрации новых строк, вычисления значения в столбце или создания столбцов с агрегатными значениями
Ordinal	Позволяет установить порядковый номер столбца в коллекции Columns в DataTable
Readonly	Определяет, будет ли столбец только для чтения (если да, то значения в этом столбце после добавления строки изменять уже будет невозможно). По умолчанию имеет значение false
Table	Возвращает DataTable, которой принадлежит данный объект DataColumn
Unique	Позволяет определить, будут ли в столбце допускаться повторяющиеся значения. Если столбец является первичным ключом, то это свойство должно иметь значение true

Создаем объект DataColumn

Чтобы показать возможности работы с DataColumn, мы смоделируем при помощи этого типа столбец FIrstName (символьного типа). Представим также, что по условиям задачи нам нужно, чтобы этот столбец был доступен только для чтения. Код для создания соответствующего объекта DataColumn может быть таким:

```
protected void btnColumn_Click (object sender. System.EventArgs e)
{
        // Создаем столбец FirstName
        DataColumn colFName = new DataColumn();

        // Настраиваем его параметры
        colFName.DataType = Type.GetType("System.String");
        colFName.ReadOnly = true;
        colFName.Caption = "First Name":
        colFName.ColumnName = "FirstName";

        // А теперь извлекаем информацию о них
        string temp = "Column type: " + colFName.DataType + "\n" + "Read only? " +
                                        colFName.ReadOnly + "\n" + "Caption: " +
                                        colFName.Caption + "\n" + "Column Name: " +
                                        colFName.ColumnName + "\n" + "Nulls allowed?
                                        " + colFName.AllowDBNull;

        MessageBox.Show(temp, "Column Properties");
}
```

Результат выполнения этого кода представлен на рис. 13.3.

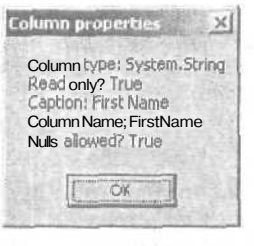


Рис. 13.3. Некоторые свойства типа DataColumn

Для типа DataColumn предусмотрено несколько перегруженных конструкторов, при помощи которых можно определять многие свойства объекта в момент его создания:

```
// Создаем столбец FirstColumn (второй заход)
DataColumn colFName = new DataColumn("FirstName", Type.GetType("System.String"));
colFName.ReadOnly = true;
colFName.Caption = "First Name";
```

Помимо рассмотренных нами свойств, в типе DataColumn также имеется набор методов, информацию о которых можно получить при помощи электронной документации к C#.

Добавляем объект DataColumn в DataTable

Как правило, столбцы существуют не сами по себе, а внутри таблицы. Чтобы поместить столбец (объект DataColumn) в таблицу, нам потребуется объект таблицы (DataTable). Создание объекта DataTable будет рассмотрено позднее в этой главе, а пока представим себе, что он у нас уже создан. Объект DataColumn добавляется во внутреннюю коллекцию DataTable.DataColumnCollection, а производится эта операция при помощи свойства DataTable.Columns:

```
// Создаем столбец myColumn
DataColumn myColumn = new DataColumn();

// Создаем таблицу myTable
DataTable myTable = new DataTable("MyTable");

// Свойство Columns возвращает объект типа DataColumnCollection.
// Для добавления столбца в таблицу используется метод Add()
myTable.Columns.Add(myColumn);
```

Делаем столбец первичным ключом таблицы

Один из основных принципов проектирования баз данных гласит: в каждой таблице обязательно должен быть первичный ключ (primary key). Ограничение первичного ключа используется для того, чтобы однозначно идентифицировать любую строку таблицы. В нашем примере с таблицей Employees первичным ключом будет столбец EmpID. Значения в столбце, определенном как первичный ключ, должны быть уникальными, кроме того, в нем не допускаются значения типа NULL (пустые значения), поэтому нам придется настроить соответствующие значения свойств AllowDBNull и Unique:

```
// Столбец EmpID будет первичным ключом таблицы
DataColumn colEmpID = new DataColumn(EmpID, Type.GetType("System.Int32"));
colEmpID.Caption = "Employee ID";
colEmpID.AllowDBNull = false;
colEmpID.Unique = true;
```

Однако это еще не все: чтобы столбец стал первичным ключом таблицы, нам еще потребуется воспользоваться свойством DataTable.PrimaryKey. Это мы сделаем чуть позже — при рассмотрении возможностей типа DataTable.

Настройка автоматического увеличения значений для столбцов

В реальных таблицах очень часто используется свойство автоинкремента — автоматического увеличения значения столбца при добавлении в таблицу новых строк. Как только мы добавляем новую строку в таблицу, в соответствующее поле этой строки будет автоматически добавлено новое значение, основанное на значении предыдущей строки и на шаге приращения. Обычно такие столбцы называются столбцами счетчика (identity columns). Такое решение позволяет создавать уникальные (в большинстве случаев) значения для строк автоматически.

При определении столбца счетчика нам потребуется настроить значения свойств AutoIncrement (определяет сам тип столбца и указывает, что значения для него нужно будет автоматически увеличивать), AutoIncrementSeed (начальное значение для столбца) и AutoIncrementStep (шаг приращения). Соответствующий код может быть таким:

```
// Создаем столбец данных
DataColumn myColumn = new DataColumn();
myColumn.ColumnName = "Foo";
myColumn.DataType - System.Type.GetType("System.Int32");

// Настраиваем автоматическое увеличение значений
myColumn.AutoIncrement = true;
myColumn.AutoIncrementSeed = 500;
myColumn.AutoIncrementStep = 12;
```

Мы определили исходное значения для столбца Foo, равное 500, а шаг приращения — 12. Первому значению в этом столбце будет присвоено значение 12, а последующим — соответственно 512, 524, 536 и т. д.

Давайте проверим, так ли это. Вставим наш столбец Foo в таблицу, затем добавим в нее несколько строк и посмотрим, что будет со значениями столбца Foo для этих строк:

```
protected void btnAutoCol_Click (object sender. System.EventArgs e)
{
        // Создаем столбец данных
        DataColumn myColumn = new DataColumn();
        myColumn.ColumnName = "Foo";
        myColumn.DataType = System.Type.GetType("System.Int32");

        // Настраиваем автоматическое увеличение значений
        myColumn.AutoIncrement - true;
        myColumn.AutoIncrementSeed = 500;
        myColumn.AutoIncrementStep = 12;

        // Добавляем этот столбец в таблицу
        DataTable myTable = new DataTable("MyTable");
        myTable.Columns.Add(myColumn);

        // Добавляем 20 строк
        DataRow r;
        for(int i = O; i < 20: 1++)
        {
                r = myTable.NewRow();
                myTable.Rows.Add(r);
```

```
        }

        // А теперь выводим значения для каждой строки
        string temp = "";
        DataRowCollection rows = myTable.Rows;
        for(int i = 0; i < myTable.Rows.Count; i++)
        {
              DataRow currRow = rows[i];
              temp += currRow["Foo"] + " ";
        }
        MessageBox.Show(temp, "These values brought ala auto-increment");
}
```

Когда мы запустим приложение и щелкнем на соответствующей кнопке, должно открыться окно, представленное на рис. 13.4.

Рис. 13.4. Столбец счетчика — автоматическое увеличение значений

Настраиваем представление столбца в формате XML

Подавляющее большинство свойств типа DataColumn, которые остались за рамками предыдущих разделов, являются вполне очевидными (особенно если вы владеете соответствующей терминологией для работы с базами данных). Однако на одном свойстве хотелось бы остановиться подробнее. Это свойство — ColumnMapping, и оно определяет, как данный столбец будет представлен в формате XML при извлечении содержимого столбца при помощи метода WriteXml(). Для свойства *ColumnMapping* используются значения из перечисления MappingType (табл. 13.4).

Таблица 13.4. Значения перечисления MappingType

Значение перечисления MappingType	Описание
Attribute	Столбцу соответствует атрибут XML
Element	Столбцу соответствует элемент XML (это значение используется по умолчанию)
Hidden	Столбцу соответствует внутренняя структура
TableElement	Столбцу соответствует значение таблицы
Text	Столбцу соответствует значение текста

По умолчанию для ColumnMapping используется значение MappingType.Element. Что же это значит? Все очень просто. Предположим, что вы записываете содержимое DataSet в текстовый файл в формате XML. При использовании значения Element столбцу EmpID в текстовом файле будут соответствовать значения вида:

```
<Employee>
      <EmpID>500</EmpID>
</Employee>
```

Если же мы установим для свойства ColumnMapping значение MappingType.Attribute, то столбцу EmpID в текстовом файле XML будут соответствовать уже следующие строки:

```
<Employee EmpID = "500"/>
```

Подробнее об интеграции ADO.NET и XML будет рассказано ниже, когда мы будем обсуждать тип DataSet. Сейчас же мы (надеемся) внесли некоторую ясность в отношении типа DataColumn и приступаем к рассмотрению еще одного очень важного типа — DataRow.

Код приложения DataColumn можно найти в подкаталоге Chapter 13.

Тип DataRow

Как мы только что убедились, структура таблицы определяется как коллекция объектов DataColumn. Для хранения этой коллекции в объекте DataTable используется внутренний объект DataColumnCollection. Помимо этого, для DataTable важна еще одна коллекция: коллекция объектов DataRow, которая и определяет собственно данные, хранящиеся в таблице. Если в нашей таблице Employees предусмотрено 20 записей для сотрудников, каждую запись будет представлять отдельный объект DataRow. При помощи членов класса DataRow мы можем производить операции вставки, изменения и удаления строк из таблицы, а также сравнивать значения, которые содержатся в строках.

Работа с DataRow несколько отличается от работы с DataColumn, поскольку объект DataRow не нужно создавать напрямую. Вместо этого мы получаем на него ссылку при помощи объекта DataTable. Например, предположим, что нам потребовалось вставить новую строку в нашу таблицу Employees. Для этого нам потребуется метод DataTable.NewRow():

```
// Создаем объект таблицы
DataTable empTable = new DataTable("Employees");

//...Считаем, что здесь мы добавили в таблицу столбцы EmpID. FirstName и LastName...

// Создаем строку для сотрудника
DataRow row = empTable.NewRow();
row["EmpID"] = 102;
row["FirstName"] = "Joe";
row["LastName"] = "Blow";

// Добавляем ее во внутреннюю коллекцию строк в таблице - DataRowCollection
empTable.Rows.Add(row);
```

Как мы видим на нашем примере, для хранения данных о строках в объекте DataTable используется еще одна внутренняя коллекция — DataRowCollection.

Теперь еще немного о типе DataRow. Наиболее важные свойства этого типа представлены в табл. 13.5. Кроме того, отметим, что класс DataRow определяет индексатор, при помощи которого можно получить значение из поля строки по порядковому номеру. Конечно же, то же самое значение можно будет получить и по имени столбца.

Таблица 13.5. Члены класса DataRow

Член	Назначение
AcceptChanges() RejectChanges()	Для записи в строку (или отказа от них) всех изменений, произведенных начиная с момента, когда последний раз был вызван метод AcceptChanges()
BeginEdit() EndEdit() CancelEdit()	Начать, завершить, прекратить операции редактирования для объекта DataRow
Delete()	Помечает строку для удаления при следующем вызове метода AcceptChanges()
HasErrors GetColumnsInErrors() GetColumnError() ClearErrors() RowError	Свойство HasErrors возвращает логическое значение, определяющее, присутствуют ли ошибки в значениях столбцов для данной строки. Если такие ошибки есть, то для получения значений, которые нарушают установленные правила, можно использовать метод GetColumnsInError(). Для получения описания ошибки можно использовать GetColumnError(), а ClearErrors() просто удаляет все ошибочные значения из строки. Свойство RowError позволяет настроить текстовое описание для ошибки в столбце
IsNull()	Возвращает информацию о том, содержит ли строка в указанном поле пустое значение (значение типа NULL)
ItemArray	Позволяет получить или установить значения всех полей строки при помощи массива объектов
RowState	Позволяет получать информацию о текущем состоянии объекта DataRow. Используются значения из перечисления RowState
Table	Это свойство используется для получения указателя на таблицу, содержащую текущий объект DataRow

Работа со свойством DataRow.RowState

Большинство методов и свойств класса DataRow используется только в контексте размещения объекта DataRow в таблице (DataTable). Мы вскоре познакомимся с операциями вставки, удаления и изменения строк, но до этого мы рассмотрим свойство RowState класса DataRow. Главное назначение этого свойства — определять (в процессе выполнения программы), в каком состоянии находятся выбранные нами строки в таблице: были ли они изменены, только что вставлены и т. п. Для свойства RowState используются значения из перечисления DataRowState (табл. 13.6).

Таблица 13.6. Значения перечисления DataRowState

Значение	Описание
Deleted	Строка была изменена при помощи метода DataRow.Delete
Detached	Строка была создана, но она еще не является частью DataRowCollection. Обычно строка находится в таком состоянии непосредственно после вставки или после принудительного вывода из коллекции DataRowCollection
Modified	Строка была изменена, но метод AcceptChanges() еще не вызывался
New	Строка была добавлена в коллекцию DataRowCollection, но метод AcceptChanges() еще не был вызван
Unchanged	Строка не была изменена с момента последнего вызова метода AcceptChanges()

Проиллюстрировать различные состояния DataRow можно при помощи следующего кода:

```
public class DRState
{
        public static void Main()
        {
                // Создаем DataTable из одного столбца
                DataTable myTable * new DataTable("Employees");
                DataColumn colID - new DataColumn("EmpID", Type.GetType("System.Int32"));
                myTable.Columns.Add(colID);

                // Начинаем работу с DataRow
                DataRow myRow;

                // Создаем объект DataRow (пока в состоянии Detached)
                myRow - myTable.NewRow();
                Console.WriteLine(myRow.RowState.ToString());

                // Теперь добавлем его в таблицу
                myTable.Rows.Add(myRow);
                Console.WriteLine(myRow.RowState.ToString());

                // Записываем изменения
                myTable.AcceptChanges();
                Console.WriteLine(myRow.RowState,ToString());

                // Изменяем строку
                myRow["EmpId"] = 100;
                Console.WriteLine(myRow.RowState.ToString());

                // А теперь ее удаляем
                myRow.DeleteO:
                Console.WriteLine(myRow.RowState.ToStringO);
                myRow.AcceptChanges();
        }
}
```

Результат работы программы комментариев не требует (рис. 13.5).

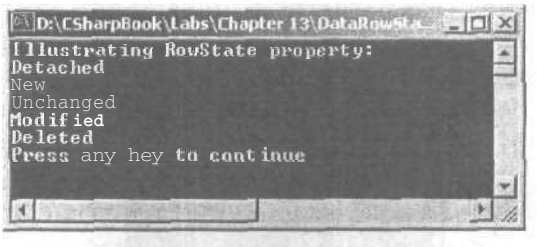


Рис. 13.5. Разные состояния строки

Как мы могли убедиться, ADO.NET позволяет отслеживать различные состояния строк. Очень важно, что при помощи свойства RowState можно выяснить, какие строки были изменены, а какие — нет. Основываясь на этой информации, ADO.NET передает по сети для записи в базу данных только те строки, которые были реально изменены пользователем, что во многих случаях экономит значи-

тельное количество сетевого трафика. Знание возможностей работы со свойством DataRow.Rowstate позволит нам оптимизировать передачу данных от одного уровня приложения другому.

Работа со свойством ItemArray

Еще один очень важный член класса DataRow — свойство ItemArray. Это свойство позволяет получить полный «снимок» текущей строки в виде массива объектов типа System.Object. Кроме того, при помощи этого свойства мы можем вставить в таблицу новую строку, не указывая явно значения для каждого столбца. Проиллюстрируем это примером. Предположим, что в нашей таблице есть два столбца — EmpID и FirstName. Мы можем добавить новые строки, передавая массив объектов через свойство ItemArray:

```
// Объявляем массив
object [] myVals = new object[2];
DataRow dr;

// Создаем новые строки и добавляем их в DataRowCollection
for (int i = 0; i < 5; i++)
{
        myVals[0] = i;
        myVals[1] = "Name " + i;
        dr = myTable.NewRow();
        dr.ItemArray = myVals;
        myTable.Rows.Add(dr);
}
// А теперь выводим каждое из значений
foreach(DataRow r in myTable.Rows)
{
        foreach(DataColumn c in myTable.Columns)
        {
                Console.WriteLine(r[c]);
        }
}
```

Результат работы программы представлен на рис. 13.6.

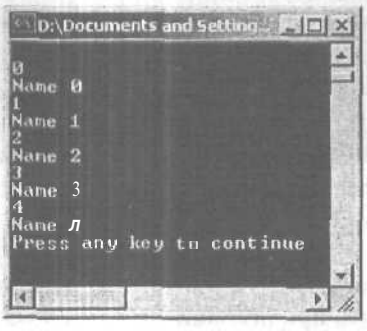


Рис. 13.6. Применение свойства ItemArray

Код приложения DataRowState можно найти в подкаталоге Chapter 13.

Тип DataTable

Класс DataTable используется для создания в оперативной памяти моделей табличных наборов данных. Мы можем создавать объекты DataTable программным образом, однако чаще в приложениях объект DataTable создается автоматически с помощью возможностей DataSet и типов, определенных в пространствах имен System.Data.OleDb и System.Data.SqlClient. Наиболее важные свойства DataTable представлены в табл. 13.7.

Таблица 13.7. Свойства класса DataTable

Свойство	Описание
CaseSensitive	Определяет, будет ли при сравнении символьных данных в таблице учитываться регистр символов. По умолчанию — false (не будет)
Child Relations	Возвращает коллекцию подчиненных отношений (DataRelationCollection) для объекта DataTable (если такие отношения есть)
Columns	Возвращает набор столбцов для таблицы
Constraints	Позволяет получить коллекцию ограничений, определенных в таблице (ConstraintCollection)
DataSet	Позволяет получить ссылку на объект DataSet, к которому принадлежит данная таблица (если такой объект есть)
DefaultView	Позволяет получить настроенное представление для таблицы, которое может включать в себя, например, только некоторые выбранные пользователем столбцы или данные о положении курсора
MinimumCapacity	Позволяет получить или установить исходное количество строк в таблице (по умолчанию используется значение 25)
ParentRelations	Позволяет получить коллекцию родительских отношений для данного объекта DataTable
PrimaryKey	Позволяет получить или установить массив столбцов, которые являются первичным ключом в таблице
Rows	Возвращает набор строк, относящихся к таблице
TableName	Позволяет получить имя таблицы или определить его. Значение для этого свойства может быть установлено через конструктор для таблицы

Графическая схема наиболее важных компонентов DataTable представлена на рис. 13.7. Обратите внимание, что схема не имеет никакого отношения к иерархии классов (к примеру, класс DataRow не является классом, производным от DataRowCollection). Эта схема представляет логические отношения «иметь» ("has-a") между наиболее важными компонентами DataTable (например, объекты DataRow принадлежат к объекту DataRowCollection).

Создаем объект DataTable

Теперь, когда мы познакомились с классом DataTable, следующая наша задача — создать полный объект DataTable и продемонстрировать возможности работы с этим объектом. Предположим, что наша цель — создать объект DataTable, представляющий список автомобилей в базе данных Cars. В нашей таблице Inventory будет четыре столбца: CarID (идентификатор автомобиля), Make (модель), Color (цвет)

и PetName (прозвище). Столбец CarID будет столбцом счетчика (то есть значения в нем будут увеличиваться автоматически), и, кроме того, он будет первичным ключом таблицы. Столбец PetName будет допускать значения типа NULL. То, как наша таблица может выглядеть, представлено на рис. 13.8.

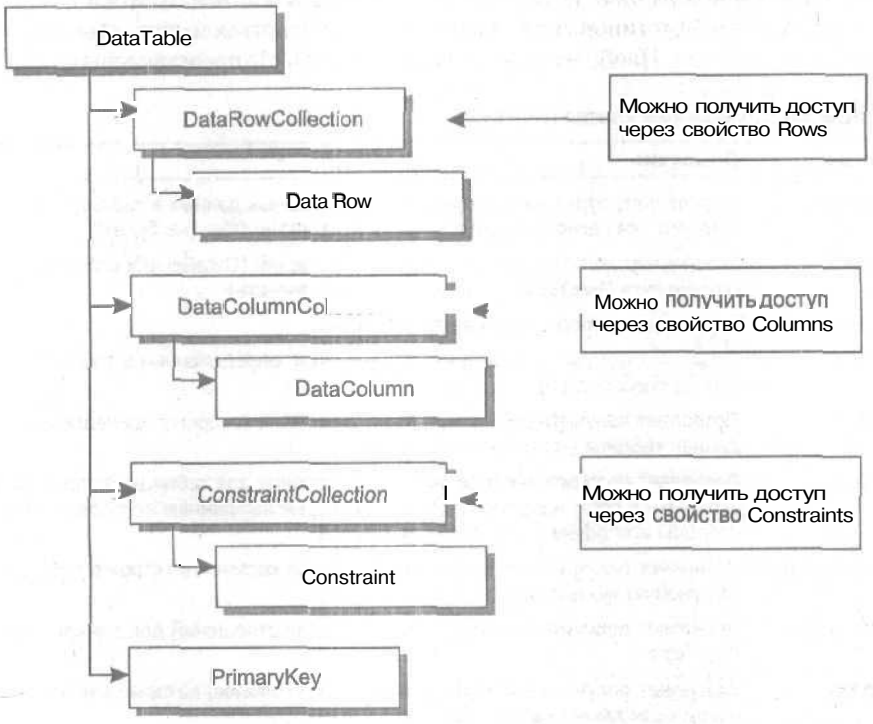


Рис. 13.7. *Отношения* компонентов DataTable

CarID (PK)	Make	Color	PetName
0	BMW	Green	Chucky
1	Yugo	White	Tiny
2	Jeep	Tan	(null)
3	Caravan	Pink	Pain Inducer

Рис. 13.8. Таблица Inventory

Первое, что нужно сделать, — создать объект DataTable. Очень удобно указать дружественное имя таблицы в качестве параметра конструктора. Это дружественное имя затем можно будет использовать, например, при обращении к нашей таблице из DataSet. Код для создания объекта DataTable в нашем случае будет выглядеть так:

```
// Создаеи объект DataTable
DataTable inventoryTable = new DataTable("Inventory");
```

Наша следующая задача — программным образом произвести добавление необходимых столбцов в коллекцию DataColumnCollection при помощи метода Add(). Нам нужно будет добавить четыре столбца: CarID, Make, Color и PetName (тип данных для каждого столбца определяется при помощи свойства DataColumn.DataType):

```
// Объявляем переменную DataColumn
DataColumn myDataColumn;

// Создаем столбец CarID и добавляем его в таблицу
myDataColumn = new DataColumn();
myDataColumn.DataType = Type.GetType("System.Int32");
myDataColumn.ColumnName = "CarID";
myDataColumn.Readonly = true;
myDataColumn.AllowDBNull = false;
myDataColumn.Unique = true;

// Настраиваем CarID как столбец счетчика
myDataColumn.AutoIncrement = true;
myDataColumn.AutoIncrementSeed = 1000;
myDataColumn.AutoIncrementStep = 10;
inventoryTable.Columns.Add(myDataColumn);

// Создаеи столбец Make и добавляем его в таблицу
myDataColumn = new DataColumn();
myDataColumn.DataType = Type.GetType("System.String");
myDataColumn.ColumnName = "Make";
inventoryTable.Columns.Add(myDataColumn);

// Создаем столбец Color и добавляем его в таблицу
myDataColumn = new DataColumn();
myDataColumn.DataType = Type.GetType("System.String");
myDataColumn.ColumnName = "Color";
inventoryTable.Columns.Add(myDataColumn);

// Создаем и добавляем последний столбец - PetName
myDataColumn = new DataColumnO;
myDataColumn.DataType = Type.GetType("System.String");
myDataColumn.ColumnName = "PetName";
myDataColumn.AllowDbNull = true;
inventoryTable.Columns.Add(myDataColumn);
```

Перед тем как приступить к добавлению строк, нам еще потребуется указать первичный ключ для таблицы. Для этого положено использовать свойство DataTable.PrimaryKey. Первичный ключ может включать в себя несколько столбцов (композитный первичный ключ), поэтому это свойство в качестве параметра принимает массив объектов DataColumn. В нашей таблице роль первичного ключа будет выполнять единственный столбец CarID, поэтому соответствующий код может выглядеть так:

```
// Определяем в качестве первичного ключа столбец CarID
DataColumn[] PK = new DataColumn[1];
PK[0] = inventoryTable.Columns["CarID"];
inventoryTable.PrimaryKey = PK;
```

Таблицу мы создали не просто так, а для того, чтобы помещать в нее данные. Будем считать, что у нас уже есть массив arTheCars (типа ArrayList) объектов Car. Перенос из него данных об автомобилях в таблицу может выглядеть так:

```
// Последовательно создаем строки на основе данных элементов массива. CarID можно
// опускать, поскольку мы настроили автонатическое приращение значений для этого столбца
foreach(Car c in arTheCars)
{
        DataRow new Row;
        newRow - inventoryTable.NewRow();
        newRow["Make"] - c.make;
        newRow["Color"] - c.color;
        newRow["PetName"] - c.petName:
        inventoryTable.Rows.Add(newRow);
}
```

Теперь наша задача — убедиться, что все прошло нормально. Чтобы не отвлекаться, вообразим, что у нас есть приложение Windows Forms, на главной форме которого размещен элемент управления DataGrid. Для того чтобы привязать DataTable к DataGrid, используется свойство DataGrid.DataSource. По выполнении всех необходимых действий мы должны получить что-то похожее на рис. 13.9.

Рис. 13.9. Отображаем содержимое DataTable при помощи элемента управления DataGrid

В нашем примере мы добавили строки, явно указывая имя каждого столбца. Однако то же самое можно сделать и по-другому: указывая вместо имени порядковый номер столбца в таблице. Код для создания строк таблицы при этом может выглядеть так (результаты работы будут совершенно одинаковы):

```
foreach(Car c in arTheCars)
{
        // Указываем столбцы, используя их порядковый номер
        DataRow new Row;
        newRow = inventoryTable.NewRow();
        newRow[1] = c.make;
        newRow[2] - c.color;
        newRow[3] = c.petName;
        inventoryTable.Rows.Add(newRow);
}
```

Удаляем строки из таблицы

Если нам потребуется удалить из таблицы одну или несколько строк, мы можем сделать это несколькими способами. Первый способ — воспользоваться ме-

тодом DeleteO. При этом нам придется указать порядковый номер удаляемой строки.

Предположим, что в графическом интерфейсе нашего приложения произошли изменения: появилась кнопка Remove (рис. 13.10).

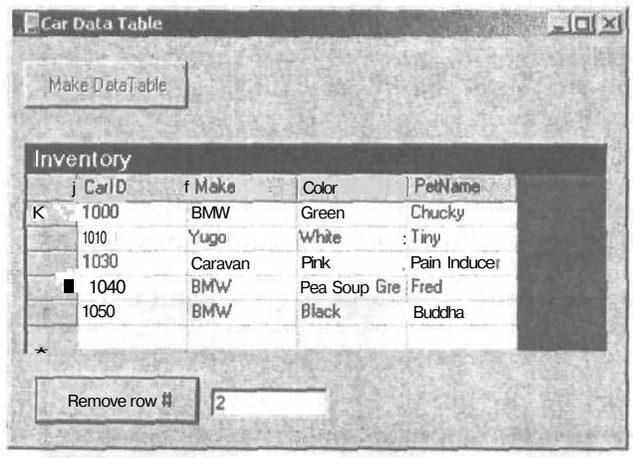


Рис. 13.10. Графическое приложение с возможностью удаления строк

Будем считать, что мы только что нажали кнопку Remove row # и строки номер *2* (CarID 1020) больше нет. Для выполнения этой нехитрой операции достаточно поместить в обработчик события Click для нашей кнопки следующий код:

```
// Удаляем строку из таблицы
protected void btnRemoveRow_Click (object sender. SystemEventArgs e)
{
        try
        {
                inventoryTable.Rows[int.Parse(txtRemove.Text)].Delete();
                inventoryTable.AcceptChanges();
        }
        catch(Exception ex)
        {
                MessageBox.Show(ex.Message);
        }
}
```

Метод Del ete() можно было бы, наверное, назвать MarkedAsDeletable(), поскольку при его вызове реального удаления строки не происходит. Вместо этого столбец лишь помечается как подлежащий удалению, а само удаление происходит при вызове метода DataTable.AcceptChanges(). Пока AcceptChanges О не вызван, мы можем даже восстановить удаленную строку:

```
// Помечаем строку как подлежащую удалению, а потом возвращаем ее к жизни
protected void btnRemoveRow_Click (object sender. System.EventArgs e)
{
        inventoryTable.Rows[txtRemove.Text.ToInt32()].Delete();

        // Что-то делаем
```

```
        // Возвращаем исходное состояние строки - значение RowState
        inventoryTable.RejectChanges();}
}
```

Применение фильтров и порядка сортировки для DataTable

Очень часто возникает необходимость получить подборку данных таблицы, основываясь на каком-либо критерии отбора (фильтре). Например, предположим, что в нашей таблице нам потребовалось извлечь только строки, относящиеся к автомобилям определенной модели (фильтр по значению столбца Make). В этой ситуации мы можем использовать метод DataTable.Select(). Еще раз обновим графический интерфейс нашего приложения, чтобы получить возможность выводить данные только об автомобилях выбранной нами модели (рис. 13.11).

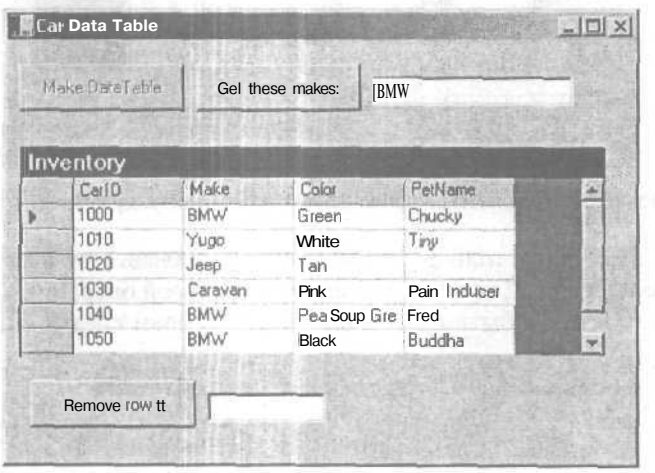


Рис. 13.11. Используем фильтр для отбора строк

Для метода Select() предусмотрено множество перегруженных вариантов. Чаще всего методу Select() передается параметр, который представляет условие для отбора строк. Например, в нашем случае для обработчика события Click кнопки отбора записей можно использовать следующий код:

```
protected void btnGetMakes_Click (object sender, System.EventArgs e)
{
        // Создаем фильтр для отбора, основываясь на значении, введенной пользователем
        string filterStr = "Make='" + txtMake.Text + "'";

        // Находим все строки, соответствующие нашему условию
        DataRow[] makes = inventoryTable.Select(filterStr);

        // А теперь выводим информацию о найденных строках
        if(makes.Length == 0)
                MessageBox.Show("Sorry, no cars...", "Selection error!");
        else
        {
```

```
        string strMake * null;
        for(int i = O; i <makes.Length; i++)
        {
            DataRow temp = makes[i];
            strMake += temp["PetName"].ToString() + "\n";
        }
        MessageBox.Show(strMake, txtMake.Text + " types(s):");
    }
}
```

Вначале мы создали текстовую строку, содержащую критерии отбора. Если бы пользователь ввел в текстовом поле BMW, то условие поиска выглядело бы как Make= 'BMW'. Затем мы передали это условие поиска методу Select() и получили в ответ массив объектов DataRow. При этом каждому из этих объектов соответствовала строка, удовлетворяющая условию поиска (рис. 13.12).

Рис. 13.12. Отобранные при помощи фильтра данные

Условие поиска может содержать в себе любое количество операторов. Например, если бы нам потребовалось найти все строки со значением CarID большим, чем 1030, то мы могли бы использовать следующий код:

```
// Выводим прозвища всех машин с CarID больше 1030
DataRow[] properIDs;
string newFilterStr = "CarID > '1030'";
properIDs = inventoryTable.Select(newFilterStr);
string strIDs = null;

for(int i = 0; 1 < properIDs.Length; 1++)
{
    DataRow temp = properIDs[i];
    strIDs += temp["PetName"].ToString() + " is ID " + temp["ID'] + "\n";
}
MessageBox.Show(strIDs, "Pet names of Cars where ID > 1030");
```

То, что должно получиться в результате выполнения программы, представлено на рис. 13.13.

Рис. 13.13. Отбираем диапазон данных

Условия поиска формулируются при помощи стандартного синтаксиса SQL. Чтобы не быть голословными, предположим, что нам нужно упорядочить возвращаемые методом Select() данные по прозвищу (в алфавитном порядке). В терминах SQL это значит, что нам необходимо провести сортировку по столбцу PetName. Ничего особо сложного нам делать не придется, поскольку для метода Select() предусмотрен перегруженный вариант и на такой случай:

```
makes = inventoryTable.Select(filterStr, "PetName");
```

Результаты запроса будут возвращены в упорядоченном виде (рис. 13.14).

Рис. 13.14. Упорядоченные данные

Если мы хотим, чтобы значения были упорядочены по убыванию, запрос можно переписать следующим образом;

```
// Возвращаем результаты в убывающем порядке
makes = inventoryTable.Select(filterStr, "PetName DESC");
```

Формально говоря, условия сортировки задаются путем указания имен столбцов, после которых может следовать ключевое слово ASC (от ascending, сортировка по возрастанию — она используется по умолчанию) или DESC (от descending, сортировка по убыванию). Если есть необходимость провести сортировку сразу по нескольким столбцам, названия этих столбцов можно перечислить через запятую.

Внесение изменений в строки

Последний очень важный вопрос, который мы должны прояснить, — а как можно вносить изменения в уже существующие строки в объекте DataTable? Наиболее простой способ — вначале воспользоваться тем же методом Select(), чтобы отобрать интересующие нас строки, а затем внести изменения в те объекты DataRow, которые вернет нам этот метод. Например, предположим, что в нашем приложении появилась еще одна кнопка, по нажатии на которую все значения BMW в столбце Make должны быть заменены на Colt. Код обработчика события Click для этой кнопки может выглядеть следующим образом:

```
// Находим все строки, которые нужно отредактировать, при помощи запроса
protected void btnChange_Click (object sender. System.EventArgs e)
{
        // Создаем фильтр для отбора строк
        string filterStr - "Make='BMW'";
        string strMake = null;

        // Получаем все строки, соответствующие условию
        DataRow[] makes - inventoryTable.Select(filterStr);
```

```
        // Заменяем в отобранных строках значения в столбце Make на "Colt"
        for(int i = 0; i < makes.Length; i++)
        {
            DataRow temp = makes[i];
            strMake += temp["Make"] + "Colt";
            makes[i] = temp;
        }
    }
```

Кроме того, в классе DataRow также предусмотрены методы BeginEdit(), EndEdit() и CancelEdit(), которые позволяют редактировать содержимое строки, отключив на время все правила проверки целостности данных, связанные с соответствующей строкой. Как только мы вызываем метод BeginEdit() для объекта DataRow, он переводится в режим редактирования. В этом режиме мы вносим необходимые изменения, а затем вызываем либо EndEdit() для сохранения внесенных изменений, либо CancelEdit() для их отмены. Например:

```
// Считаем, что вы уже получили объект DataRow для внесения изменений. Теперь мы
// переводим его в режим редактирования
rowToUpdate.BeginEdit();

// Передаем эту строку вспомогательной функции, возвращающей true
// или false
if(ChangeValuesForThisRow(rowToUpdate))
{
    rowToUpdate.EndEdit(); // OK!
}
else
{
    rowToUpdate.CancelEdit(); // Нет, спасибо.
}
```

Методы BeginEdit(), EndEdit() и CancelEdit() можно вызывать вручную, но чаще они вызываются автоматически в процессе редактирования строк через элемент управления DataGrid, связанный с DataTable. Например, если мы выбрали строку для внесения изменений в DataGrid, то эта строка автоматически переводится в режим редактирования. При перемещении фокуса на новую строку автоматически вызывается метод EndEdit(). Проверить это можно очень просто: заменим значения столбца Make на BMW для всех автомобилей через элемент управления DataGrid нашего приложения (рис. 13.15), а затем точно так же запросим все строки со значением BMW. Нам должны вернуться все строки, в которые мы внесли через DataGrid соответствующие изменения (рис. 13.16).

Тип DataView

В мире баз данных представление (view) — это специальным образом настроенное отображение данных из таблицы (или нескольких таблиц). Например, мы можем создать представление на основе нашей таблицы Inventory, которое будет отображать только автомобили определенного цвета. В ADO.NET представления (объекты DataView) позволяют программным образом получать выборку данных на основе базовой таблицы (объекта DataTable).

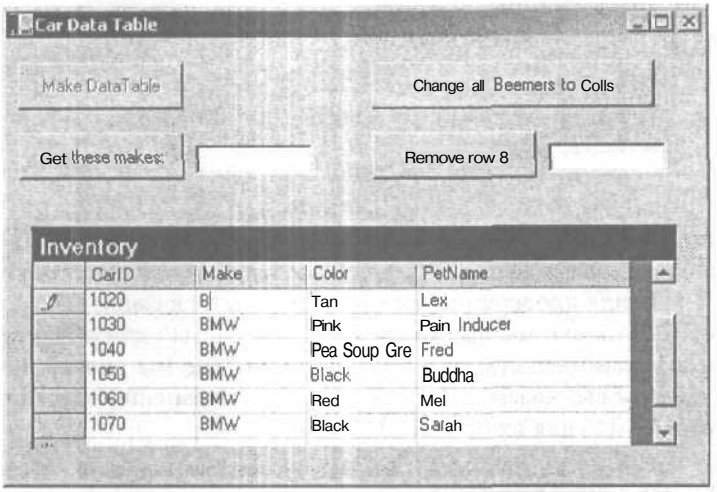


Рис. 13.15. Редактируем строки с помощью DataGrid...

Рис. 13.16. ...а затем получаем их с помощью запроса к DataTable

Для одной и той же таблицы мы можем создать неограниченное число представлений. Создание нескольких представлений для одной таблицы может потребоваться, например, если мы планируем использовать эти представления в разных элементах управления (например, если у нас на форме размещено несколько элементов управления DataGrid). Например, первый DataGrid может быть привязан к объекту DataView, который позволяет получить строки для всех автонобилей в таблице, второй DataGrid работает через DataView, выводящий информацию только о зеленых автомобилях и т. п. В классе DataTable предусмотрено свойство DefaultView, при помощи которого можно получить представление по умолчанию для данной таблицы.

Рассмотрим это на примере. Пусть на главной форме нашего приложения теперь будет размещен не один элемент управления DataGrid, а целых три. Первый, как и раньше, будет выводить информацию обо всех автомобилях в DataTable, второй — только автомобили красного цвета (значения в столбце Color равны Red), а третий — только автомобили, для которых в столбце Make значится Colt. Выглядеть все это может так, как показано на рис. 13.17.

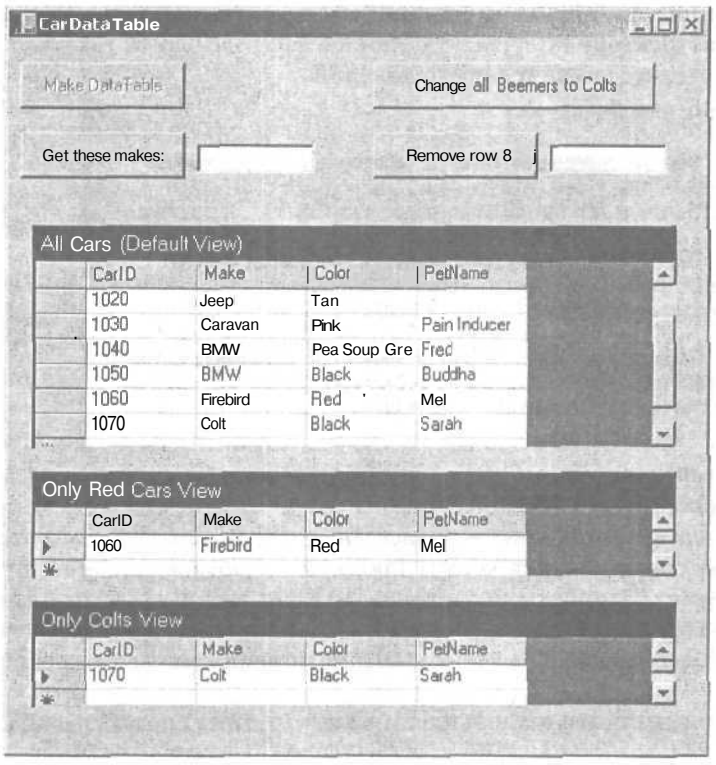


Рис. 13.17. Несколько элементов DataGrid, каждый из которых основан на своем DataView

Чтобы все это реализовать на практике, первое, что мы должны сделать — объявить необходимые переменные класса DataView:

```
public class mainForm : System.Windows.Forms.Form
{
        // Переменные для наших представлений
        DataView redCarsView;        // Только красное
        DataViewcoltsView;           // Только Colt
```

Г

Предположим, что в нашем распоряжении есть вспомогательная функция Create Views(), которое вызывается непосредственно после создания объекта DataTable:

```
protected void btnMakeDataTable_Click (object sender. System.EventArgs e)
{
        // Создаем объект DataTable
        MakeTableC);

        // Создаем представления
        CreateViews();
        ...
}
```

Реализация самой функции CreateViews() представлена ниже. Обратите внимание, что конструктору DataView передается в качестве параметра объект DataTable, на основе которого создается представление:

```
private void CreateViews()
{
        // Вначале определяем базовую таблицу для представлений
        redCarsView = new DataView(inventoryTable);
        coltsView = new DataView(inventoryTable);

        // Теперь настраиваем представления при помощи фильтра
        redCarsView.RowFilter - "Color - 'Red'";
        coltsView.RowFilter - "Make = 'colt'";

        // Привязываем представления к элементам управления DataGrid
        RedCarViewGrid.DataSource = redCarsView:
        ColtsViewGrid.DataSource • coltsView:
}
```

Как мы могли видеть, в классе DataView предусмотрено свойство RowFilter, для которого используется текстовая строка с критериями отбора объектов DataRow. После того как объекты DataView полностью сконфигурированы, мы просто указываем их в качестве источника данных для элемента управления DataGrid (через свойство DataSource). Элементы управления OataGrld умеют отслеживать изменения, которые вносятся в источник данных для них. Поэтому если мы заменим в DataTable все BMW на Colt, то значения в DataGrid автоматически изменятся, отобразив эти перемены.

Некоторые наиболее важные члены класса DataView представлены в табл. 13.8.

Таблица 13.8. Члены класса DataView

Член	Назначение
AddNew()	Добавляет новую строку через DataView
AllowDelete AllowEdit AllowNew	Позволяют настроить возможность проведения через DataView операций по удалению, редактированию и добавлению новых строк в таблицу
Delete()	Позволяет удалить существующую строку через DataView (необходимо указать ее порядковый номер)
RowFilter	Позволяет получить или установить выражение, используемое для фильтрации строк, отображаемых через DataView
Sort	Позволяет настроить порядок сортировки строк в DataView
Table	Позволяет получить или указать базовую таблицу для DataTable

Код приложения CarDataTable можно найти в подкаталоге Chapter 13.

Возможности класса DataSet

К этому моменту мы с вами уже умеем создавать таблицы с данными (объекты DataTable) в оперативной памяти и выполнять с ними все необходимые операции. В принципе, это может быть вполне достаточно для многих приложений. Однако гораздо чаще таблицы в базах данных используются не сами по себе, а во взаимо-

действии с другими таблицами. В ADO.NET возможности работы с наборами таблиц, взаимосвязанных друг с другом, предоставляет класс DataSet. В действительности при использовании объектов для доступа к источникам данных, поставляемых с ADO.NET, нам чаще всего не удастся получить объект DataTable с данными напрямую — все интерфейсы для доступа к данным в ADO.NET возвращают именно объект DataSet (как набор взаимосвязанных объектов DataTable).

Объект DataSet — это создаваемый в оперативной памяти набор таблиц (объектов DataTable), связанных между собой отношениями и снабженными средствами проверки целостности данных (для них в DataSet предусмотрены свои объекты). Разобраться в иерархии классов, входящих в DataSet, поможет схема на рис. 13.18.

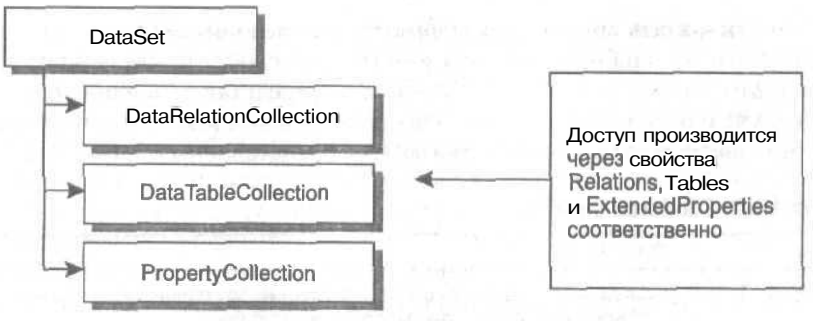


Рис. 13.18. Внутренние коллекции DataSet

Через свойство Tables объекта DataSet мы можем получить доступ к отдельным объектам DataTable, которые хранятся в коллекции DataTableCollection. Через другое свойство — Relations, мы можем получить доступ к объектам DataRelation, которые хранятся в коллекции DataRelationCollection. Поскольку DataSet — это фактически представление базы данных, которое помещается в оперативную память клиента (и с которой можно работать при разорванном соединении), то в этой модели объекты DataRelation представляют отношения между таблицами базы данных.

Например, предположим, что в одной таблице существует внешний ключ, ссылающийся на первичный ключ в другой таблице. Это отношение представлено объектом DataRelation (и его можно добавить в DataSet при помощи свойства DataSet.Relations). После этого мы можем использовать это отношение при выполнении запросов к таблицам. Всеми этими операциями нам предстоит еще заняться в этой главе.

Свойство ExtendedProperties обеспечивает доступ к объектам, хранящимся в коллекции PropertyCollection. Основное назначение всей этой конструкции со свойством ExtendedProperties и коллекцией PropertyCollection — обеспечить возможность ассоциировать дополнительную информацию (в виде пар имя — значение) с объектом DataSet. В принципе этой дополнительной информацией может быть все, что угодно. Например, мы можем самостоятельно определить, что у нашего объекта DataSet будет свойство CompanyName, а в качестве значения этого свойства указать имя нашей компании. Выглядеть это будет так:

```
// Создаем объект DataSet и добавляем для него некоторую дополнительную информацию
// (метаданные)
```

```
DataSet ds = new DataSet("MyDataSet");
ds.ExtendedProperties.Add("CompanyName", "InterTech, Inc");

// Выводим информацию о только что созданных метаданных
Console.WriteLine(ds.ExtendedProperties["CompanyName"].ToString());
```

Мы можем использовать расширенные свойства для хранения самой разной информации — например, о внутреннем пароле для доступа к данным, об интервале синхронизации данных и т. п. Кроме того, расширенные свойства (то есть свойство ExtendedProperties) предусмотрены не только для DataSet, но и DataTable.

Члены класса DataSet

Перед тем как начать вникать в особенности применения DataSet в приложении, мы рассмотрим его наиболее важные свойства. Эти свойства обеспечивают доступ к внутренним коллекциям DataSet, позволяют представлять данные из DataSet в формате XML и обеспечивают возможность получения информации об ошибках. Некоторые наиболее важные свойства DataSet представлены в табл. 13.9.

Таблица 13.9. Свойства DataSet

Свойство	Описание
CaseSensitive	Определяет, будет ли во время операций по сравнению текстовых строк в объектах DataTable учитываться регистр букв
DataSetName	Позволяет получить или задать имя для данного объекта DataSet. Обычно значение этого свойства задается как параметр, передаваемый конструктору
DefaultViewManager	Позволяет определить представление по умолчанию для отображения данных в DataSet
EnforceConstraints	Позволяет отключить (или включить снова) проверку соответствия ограничениям при выполнении операций обновления данных в DataSet
HasErrors	Позволяет получить значение, определяющее наличие ошибок в DataSet (то есть ошибок в любой строке любой таблицы DataSet)
Relations	Позволяет обратиться к коллекции отношений между таблицами DataSet
Tables	Позволяет получить доступ к коллекции таблиц DataSet

Многие методы DataSet дублируют возможности, которые обеспечиваются свойствами. Помимо взаимодействия с потоками данных в формате XML, методы DataSet позволяют копировать содержимое DataSet, устанавливать начало и конец пакетных изменений данных в DataSet и т. п. Самые важные методы DataSet представлены в табл. **13.10.**

Таблица 13.10. Методы DataSet

Метод	Описание
AcceptChanges()	Позволяет сохранить в DataSet все изменения, произведенные с момента последнего вызова этого метода
Clear()	Полная очистка DataSet — удаляются все строки из всех таблиц
Clone()	Клонирует структуру DataSet, включая структуру таблиц, отношения между таблицами и ограничения

Метод	Описание
Copy()	Копирует DataSet (структуру вместе с данными)
GetChanges()	Возвращает копию DataSet, которая содержит все изменения, внесенные в оригинальный DataSet с момента последнего вызова для него метода AcceptChanges()
GetChildRelations()	Возвращает коллекцию подчиненных отношений для указанной таблицы
GetParentRelations()	Возвращает коллекцию родительских отношений для указанной таблицы
HasChanges()	Этот перегруженный метод позволяет получить информацию об изменениях, внесенных в DataSet (отдельно по вставленным, удаленным и измененным строкам)
Merge()	Этот перегруженный метод позволяет производить слияние разных объектов DataSet
ReadXml() ReadXmlSchema()	Позволяют считывать данные в формате XML в DataSet из потока (файла, оперативной памяти, сетевого ресурса)
RejectChanges()	Отменяет все изменения, внесенные в DataSet с момента его создания или последнего вызова AcceptChanges()
WriteXml() WriteXmlSchema()	Позволяют записать данные в формате XML из DataSet в поток

Создание объекта DataSet

Чтобы проиллюстрировать применение DataSet на практике, мы создадим специальное приложение Windows Forms. В этом приложении будет использоваться объект DataSet с тремя внутренними таблицами (объектами DataTable) — Inventory, Customers и Orders. В каждой таблице будет свой первичный ключ, при этом система первичных и внешних ключей таблиц позволит нам использовать объекты DataRelation для моделирования отношений между таблицами. Общая структура базы данных, которую мы должны реализовать при помощи объекта DataSet, представлена на рис. 13.19.

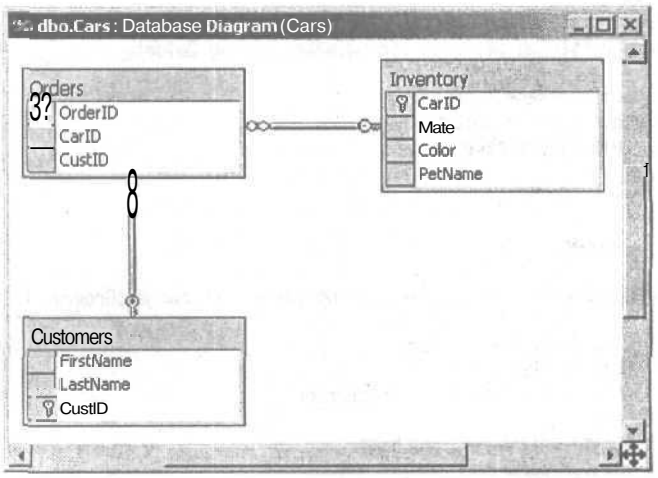


Рис. 13.19. Структура базы данных Automobile

Таблица Inventory является родительской таблицей для таблицы Orders, и для первичного ключа (столбца CarID) в таблице Inventory предусмотрен внешний ключ в таблице Orders. Таблица Customers также является родительской для таблицы Orders, и эти таблицы связаны отношением первичный/внешний ключ по столбцу CustID. Как мы увидим чуть позже, добавление в DataSet объектов DataRelation, моделирующих эти отношения, позволит нам получать взаимосвязанные данные для таблиц.

Создание объекта DataSet мы начнем с объявления переменных, представляющих таблицы и сам объект DataSet. Выглядеть это будет так:

```
public class mainForm: System.Windows.Forms.Form
{
        // Таблица Inventory
        private DataTable inventoryTable - new DataTable("Inventory");

        // Таблица Customers
        private DataTable customersTable - new DataTable("Customers");

        // Таблица Orders
        private DataTable ordersTable - new DataTable("Orders");

        // Наш DataSet
        private DataSet carsDataSet - new DataSet("CarDataSet");
...
}
```

Теперь мы создадим (чтобы удобнее было все показывать) два очень простых класса — Car и Customer, для хранения информации о которых, в сущности, и будет использован наш объект DataSet. Обратите внимание, что в классе Customer предусмотрена специальная переменная для хранения информации об автомобиле, в покупке которого заинтересован этот покупатель:

```
public class Car
{
        // Переменные мы объявляем как public исключительно для упрощения доступа
        // к ним
        public string petName. make. color;
        public Car (string petName. string make. string Color)
        {
                this.petName - petName;
                this.color - color;
                this.make - make:
        }
}

public class Customer
{
        public Customer (string fName. string lName. int currentOrder)
        {
                this.firstName - fName;
                this.lastName - lName:
              • this.currCarOrder - currentOrder;
        }
        public string firstName. lastName;
        public  int  currCarOrder;
}
```

На главной форме у нас также будут использованы два массива (объекты ArrayList) для хранения информации об автомобилях и заказчиках. Будем считать, что эти два массива будут заполняться данными в конструкторе для формы. Кроме того, в конструкторе будет производиться вызов нескольких вспомогательных функций для создания таблиц и отношений между ними. И еще одну операцию мы будем производить в конструкторе: привязывать объекты DataTable для таблиц Inventory и Customer к соответствующим элементам управления DataGrid. Подобная привязка через объект DataSet производится при помощи метода SetDataBinding() с параметрами:

```
// Списки автонобилей и заказчиков
private ArrayList arTheCars, arTheCustomers;

public mainForm()
{
        // Заполняем массив автонобилей соответствующими объектами
        arTheCars = new ArrayList();
        arTheCars.Add(new Car("Chucky", "BMW", "Green"));
        ...

        // Производим ту же операций с массивом заказчиков
        arTheCustomers = new ArrayList();
        arTheCustomers.Add(new Customer("Dave", "Brenner", 1020));
        ...

        // Создаем объекты DataTable (самым обычным способом, рассмотренным в предыдущих
        // разделах)
        MakeInventoryTable();
        MakeCustomerTable();
        MakeOrderTable();

        // Создаем объект DataRelation (об этом чуть позже)
        BuildTableRelationship();

        // Привязываем две таблицы к элементам управления DataGrid (первый параметр -
        // DataSet, второй - таблица в DataSet)
        CarDataGrid.SetDataBinding(carsDataSet, "Inventory");
        CustomerDataGrid.SetDataBinding(carsDataSet, "Customers");
}
```

Все объекты DataTable будут создаваться точно так же, как это производилось в предыдущих разделах этой главы. Чтобы проще было сосредоточиться на логике DataSet, мы опустим основную часть кода для создания таблиц. Однако отметим, что в каждой таблице у нас предусмотрен столбец счетчика, который является первичным ключом таблицы. Наиболее важная часть кода по созданию объектов DataTable представлена ниже:

```
private void MakeOrderTableO
{
...
        // Добавляем таблицу в DataSet
        cars.DataSet.tables.Add(customerstable);

        // Создаем столбцы OrderID, CustID и CarID, а затем добавляем их в таблицу
        ...
        // Назначаем столбец CarID первичным ключом
        ...
```

```
        // Добавляем несколько заказов
        for(int 1 = 0; i < arTheCustomers.Count; i++)
        {
                DataRow newRow;
                newRow = ordersTable.NewRow();
                Customer c = (Customer)arTheCustomers[i];
                newRow["CustID"] • i;
                newRow["CarID"] = c.currCarOrder;
                carsDataSet.Tables["Orders"].Rows.Add(newRow);
        }
}
```

Вспомогательные функции MakeInventoryTable() и MakeCustomerTable() в нашем случае будут создаваться точно по таким же принципам.

Моделируем отношения между таблицами при помощи класса DataRelation

Самая интересная из всего набора вспомогательных функций в конструкторе формы — это, безусловно, функция BuildTableRelationShip(), которая ответственна за создание отношений между таблицами OataSet. После того как в DataSet появилось несколько объектов таблиц, мы можем программным образом определить отношения между этими таблицами. Конечно, создание таких отношений не является обязательным, но во многих случаях это предоставляет нам важные дополнительные возможности. Например, мы сможем переходить между строками разных таблиц «на лету» и осуществлять запросы сразу к нескольким таблицам.

Объектно-ориентированную оболочку вокруг отношений между таблицами представляет класс System.Data.DataRelation. При создании объекта этого класса нам необходимо будет указать дружественное имя этого объекта, а также родительскую таблицу (например, Inventory) и подчиненную таблицу (Orders). Чтобы отношение было успешно установлено, в каждой из таблиц должен быть столбец с одинаковым названием (CarID) и типом данных (в нашем случае Int32), Полное определение функции BuildTableRelationShip() приведено ниже:

```
private void BuildTableRelationShip()
{
        // Создаем объект DataRelation
        DataRelation dr - new DataRelation("CustomerOrder";
        // Родительская таблица
        carsDataSet.Tables["Customers"].Columns["CustID"]);
        // Подчиненная таблица
        carsDataSet.Tables["Orders"].Columns["CustID"];

        // Добавляем объект DataRelation в DataSet
        carsDataSet.Relations.Add(dr);

        // Создаем еще один объект DataRelation
        dr = new DataRelation("InventoryOrder",
        // Родительская таблица
        carsDataSet.Tables["Inventory"].Columns["CarID"]);
        // Подчиненная таблица
        carsDataSet.Tables["Orders"].Columns["CarID"];
```

```
        // Добавляем и этот объект DataRelation в DataSet
        carsDataSet.Relations.Add(dr);
}
```

Объекты DataRelation хранятся в коллекции DataRelationCollection, поддерживаемой DataSet. В типе DataRelation предусмотрены свойства, которые позволяют получать ссылки на родительскую и подчиненную таблицу, участвующую в отношении, определять имя отношения и т. п. Наиболее часто используемые свойства представлены в табл. 13.11.

Таблица 13.11. Свойства типа DataRelation

Свойство	Описание
ChildColumns ChildKeyConstraint ChildTable	Позволяют получить информацию о подчиненной таблице, участвующей в отношении, а также ссылку на саму эту таблицу
DataSet	Позволяет получить ссылку на объект DataSet, к которому принадлежит данное отношение
ParentColumns ParentKeyConstraint ParentTable	Позволяют получить информацию о родительской таблице, участвующей в отношении, а также ссылку на саму эту таблицу
RelationName	Позволяет получить или задать имя для данного отношения

Переход между таблицами, участвующими в отношении

Объект DataRelation позволяет осуществлять программным образом переход между двумя таблицами, участвующими в отношении. Проиллюстрируем это на примере. Пусть на главной форме нашего приложения появятся еще два элемента управления: кнопка и текстовое поле. В текстовое поле пользователь сможет вводить идентификатор заказчика, а при нажатии на кнопку будет выводиться информация о заказе, сделанном данным заказчиком. Вывод информации для простоты будет производиться через обычный MessageBox так, как показано на рис. 13.20.

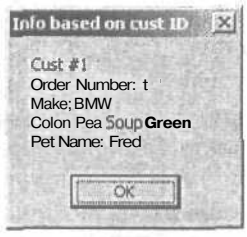


Рис. 13.20. Переход между таблицами с использованием объектов DataRelation

Код обработчика события Click для нашей кнопки может быть таким (конечно, следовало бы реализовать проверку ошибок, но для простоты мы этим заниматься не будем):

```
protected void btnGetInfo_Click (object sender. System.EventArgs e)
{
        string strInfo - "";
        DataRow drCust = null;
```

```
        DataRow[] drsOrder = null;

        // Получаем CustID из текстового поля
        int theCust - int.Parse(this.txtCustID.Text);

        // Теперь, основываясь на этом CustID, получаем всю строку из таблицы
        // Customers
        drCust - carsDataSet.Tables["Customers"].Rows[theCust];
        strInfo += "Cust #" + drCust["CustID"].ToString() + "\n";

        // Теперь производим переход от таблицы Customers в таблицу Orders
        drsOrder - drCust.GetChildRows(carsDataSet.Relations["CustomerOrder"]);

        // Получаем имя заказчика
        foreach(DataRow r in drsOrder)
              strInfo += "Order Number: " + r["OrderID"] + "\n";

        // Теперь переходим от таблицы Orders к таблице Inventory
        DataRow[] drsInv -
                    drsOrder[0].GetParentRows(carsDataSet.Relations["InventoryOrder"]);

        // Получаем информацию об автомобилях
        foreach(DataRow r in drsInv)
        {
              strInfo +- "Make: " + r["Make"] + "\n";
              strInfo += "Color: " + r["Color"] + "\n";
              strInfo +- "PetName: " + r["PetName"] + "\n";
        }
        MessageBox.Show(strInfo, "Info based on cust ID");
  }
```

Как мы можем убедиться, перемещение между таблицами производится при помощи методов, определенных в типе DataRow. Проанализируем наш код шаг за шагом. Первое, что мы сделали — получили из текстового поля значение, означающее номер заказчика (CustID), в информации о котором заинтересован пользователь. Далее мы, используя полученное значение CustID, извлекли из таблицы Customers (при помощи свойства Rows) всю строку, в которой присутствует указанное значение CustID, целиком:

```
// Получаем CustID из текстового поля
int theCust = int.Parse(this.txtCustID.Text);

// Теперь, основываясь на этом CustID. получаем всю строку из таблицы Customers
drCust = carsDataSet.Tables["Customers"].Rows[theCust];
strInfo += "Cust #" + drCust["CustID"].ToString() + "\n";
```

Следующее наше действие — переход из таблицы Customers в таблицу Orders, для чего используется отношение (объект DataRelation) CustomerOrder. Обратите внимание, что метод DataRow.GetChildRows() позволяет считывать строки из подчиненной таблицы. Получив эти строки, мы извлекаем из них нужную нам информацию:

```
// Теперь производим переход от таблицы Customers в таблицу Orders
DataRow[] drsOrder - null;
drsQrder = drCust.GetChildRows(carsDataSet.Relations["CustomerOrder"]);

// Получаем имя заказчика
foreach(DataRow r in drsOrder)
strInfo += "Order Number: " + r["OrderID"] + "\n";
```

Последний этап в нашей программе — это переход от таблицы Orders к родительской таблице Inventory при помощи метода GetParentRows(). При этом мы получаем информацию из таблицы Inventory для столбцов Make, PetName и Color:

```
// Теперь переходим от таблицы Orders к таблице Inventory
DataRow[] drsInv = drsOrder[0].GetParentRows(carsDataSet.Relations["InventoryOrder"]);

// Получаем информацию об автомобилях
foreach(DataRow r in drsInv)
{
    strInfo += "Make: " + r["Make"] + "\n";
    strInfo += "Color: " + r["Color"] + "\n";
    strInfo += "PetName: " + r["PetName"] + "\n";
}
```

Приведем еще один пример, посвященный переходам между таблицами программным образом. Код, который приведен ниже, позволяет выводить значения из таблицы Orders при помощи отношения InventoryOrders:

```
protected void btnGetChildRels_Click (object sender, System.EventArgs e)
{
        // Получаем информацию о таблицах, подчиненных по отношению к таблице
        // Inventory
        DataRelationCollection relCol;
        DataRow[] arrRows;
        string info = "";
        relCol = carsDataSet.Tables["Inventory"].ChildRelations;

        info += "\tRelation is called: " + relCol[0].RelationName + "\n\n";
        // А теперь при помощи цикла проходим по всем отношениям и выводим о них
        // информацию:
        foreach(DataRelation dr in relCol)
        {
                foreach(DataRow r in inventoryTable.Rows)
                {
                        arrRows = r.GetChildRows(dr);

                        // Выводим значения каждого столбца в строке
                        for (int i = 0; i < arrRows.Length; i++)
                        {
                                foreach(DataColumn dc in arrRows[i].Table.Columns)
                                {
                                        info += "\t" + arrRows[i][dc];
                                }
                                info += "\n";
                        }
                }
                MessageBox.Show(info, "Data in Orders Table obtained by child relations");
        }
}
```

Результат работы программы представлен на рис. 13.21.

Как мы смогли убедиться, DataSet предоставляет нам возможность работы с представлением базы данных в оперативной памяти точно так же, как и с обычной базой данных; мы можем производить запросы к таблицам, выполнять вставку, изменение и удаление данных, перемещаться между таблицами при помощи отношений между ними и т. п. Однако мы до сих пор не рассмотрели самое важное — как нам получить объект DataSet из реальной базы данных, хранящейся где-нибудь на сервере баз данных! Перед тем как перейти к рассмотрению этой темы,

мы познакомимся с еще одним исключительно важным моментом: как представлять данные из DataSet в формате XML.

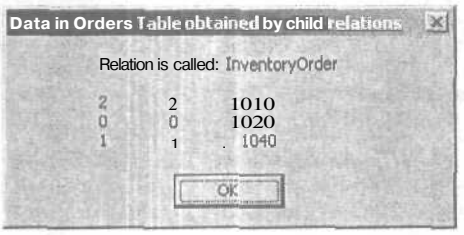


Рис. 13.21. Перемещение между таблицами при помощи отношений

Чтение и запись объектов DataSet в формате XML

Одна из главных целей, которая преследовалась при разработке ADO.NET, — это упрощение работы с данными в формате XML. При помощи типа DataSet мы можем представлять наши таблицы (со всеми данными), отношения между ними, ограничения и все остальное в формате XML и записывать их в поток (например, текстовый файл). Если у нас уже создан объект DataSet, то для записи его содержимого в формате XML достаточно просто вызывать метод WriteXml():

```
protected void btnToXML_Click (object sender. System.EventArgs e)
{
        // Записываем DataSet весь целиком в файл в каталоге app
        carsDataSet.WriteXml("cars.xml");
        MessageBox.Show("Wrote CarDataSet to XML file in app directory");
        btnReadXML.Enabled = true;
}
```

Открыв этот файл в Visual Studio.NET (рис. 13.22), можно убедиться, что весь объект DataSet (то есть вся база данных) записан в формате XML.

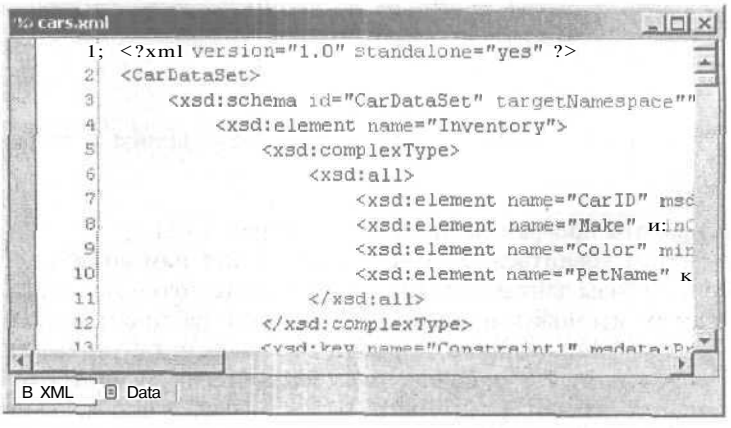


Рис. 13.22. Представление объекта DataSet в формате XML

Считывание текстовых данных из текстового файла в формате XML в объект DataSet производится при помощи метода ReadXml(). Чтобы убедиться в том, что это действительно так, проведем эксперимент. Давайте создадим в приложении специальную кнопку, при нажатии на которую DataSet вначале будет полностью очищаться, а затем восстанавливаться на основе информации из только что созданного файла в формате XML. Код для этой кнопки может выглядеть так:

```
protected void btnReadXML_Click (object sender. System.EventArgs e)
{
        // Очищаем и удаляем имеющийся объект DataSet
        carsDataSet.Clear();
        carsDataSet.Dispose();
        MessageBox.Show("Just cleared data set...");
        carsDataSet = new DataSet("CarDataSet");

        carsDataSet.ReadXml("Cars.xml");

        MessageBox.Show("Reconstructed data set from XML file...");
        btnReadXML.Enabled = false:

        // Настраиваем привязки к элементам управления DataGrid
        CarDataGrid.SetDataBinding(carsDataSet, "Inventory");
        CustomerDataGrid.SetDataBinding(carsDataSet, "Customers");
}
```

Если поглубже исследовать то, что делают WriteXml() и ReadXml C J, то выяснится, что они работают с типами, определенными в сборке System.Xml.dll (конкретно с классами XmlWriter и XmlReader). Поэтому нам потребуются, во-первых, ссылка на эту сборку, а во-вторых, указание на использование соответствующего пространства имен:

```
// Указание на использование пространства имен необходимо для вызова методов ReadXmlO
// и WriteXml()
using System.Xml;
```

Окончательный вид нашего приложения представлен на рис. 13.23.

Код приложения CarDataSet можно найти в подкаталоге Chapter 13.

Создание примера простой базы данных

Теперь, когда мы уже умеем работать с объектом DataSet, нам осталось научиться создавать его при помощи соединений с реальными базами данных.

Оставшаяся часть этой главы будет посвящена подключению к базе данных, которая потребуется нам для наших примеров. База данных включена в примеры настоящей главы и может быть создана в двух вариантах.

Первый вариант — это скрипт SQL, который позволяет создать базу данных на сервере Microsoft SQL Server версии 7.0 и последующих. Для создания базы данных откроем окно утилиты Query Analyzer (поставляемой вместе с SQL Server) и подключимся к своему серверу. Далее откроем в Query Analyzer скрипт и откорректируем его, указав пути к файлам базы данных и журнала транзакций, которые соответствуют конфигурации дисков на нашем компьютере. Части скрипта, которые, возможно, придется корректировать, выделены в приведенном ниже коде полужирным шрифтом:

```
CREATE DATABASE [Cars] ON (NAME - N'Cars_Data', FILENAME =N'
                    D:\MSSQL7\Data\Cars_Data.MDF', SIZE - 2; FILEGROWH = 10%)
LOG ON (NAME = M'Cars_Log', FILENAME - N' D:\MSSQL7\Data\Cars_Log.LDF', SIZE = 1;
FILEGROWH = 10%)
GO
```

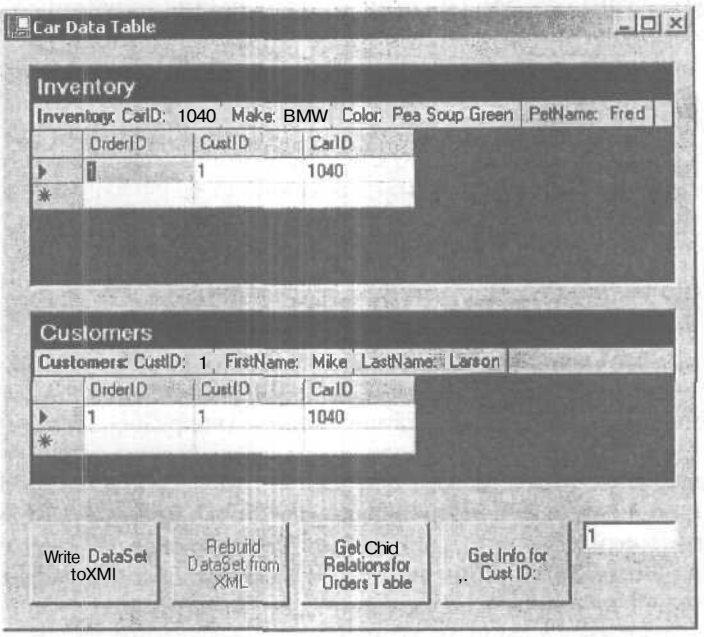


Рис. 13.23. Приложение CarDataSet: окончательный вид

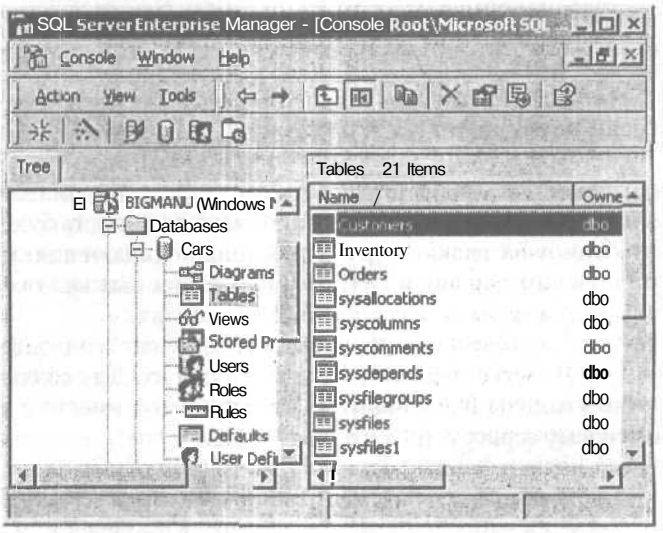


Рис. 13.24. База данных Cars на SQL Server

После этого запустим скрипт на выполнение. Чтобы убедиться, что база данных Cars со всем содержимым действительно создана, откроем SQL Server Enterprise Manager (рис. 13.24), подключимся к своему серверу, найдем в списке баз данных Cars и откроем контейнер Tables. Мы увидим среди служебных таблиц базы данных три пользовательские таблицы (Customers, Inventory и Orders) — см. рис. 13.24. Эти таблицы связаны между собой отношениями и в них загружены данные, которые потребуются нам в примерах.

Второй вариант базы данных Cars даже не нужно создавать. Он представлен в виде базы данных Microsoft Access (cars.mdb) и также находится среди файлов примеров этой главы. Структура и пользовательские данные в этой базе данных — точно такие же, как и в базе данных SQL Server, которую мы создали при помощи скрипта. В этой главе мы в основном будем работать с базой данных на SQL Server, однако методы подключения к базе данных Access также будут нами рассмотрены.

Управляемые провайдеры ADO.NET

Можно сказать, что управляемый провайдер (managed provider) в ADO.NET — это аналог провайдера OLE DB в классическом ADO. Другими словами, управляемый провайдер — это шлюз к хранилищу данных (например, на сервере баз данных), при помощи которого можно произвести загрузку данных из этого внешнего хранилища в объект DataSet.

Как уже говорилось в начале этой главы, вместе с ADO.NET поставляются два управляемых провайдера.

Первый из них — это провайдер OLE DB, который реализуется при помощи типов, определенных в пространстве имен System.Data.OleDb. Провайдер OleDb позволяет нам обращаться к данным, расположенным в любом хранилище, к которому можно подключиться по протоколу OLE рв. Таким образом, аналогично классическому ADO, вы можете использовать в ADO.NET управляемый провайдер OLE DB для доступа, например, к базам данных SQL Server, MS Access и Oracle. Однако поскольку при этом типы из пространства имен System.Data.OleDb должны взаимодействовать с обычным (не .NET) кодом драйверов OLE DB, то при таком обращении будет производиться множество преобразований вызовов .NET в вызовы COM, что в некоторых ситуациях может привести к падению производительности.

Другой управляемый провайдер — провайдер SQL — предлагает уже прямой доступ к хранилищам данных, при котором производительность будет максимальной. Однако с его помощью можно обращаться только к базам данных на MS SQL Server 7.0 и последующих версий, и только к ним. Типы, используемые провайдером SQL, содержатся в пространстве имен System.Data.SqlClient. Функциональные возможности обоих управляемых провайдеров практически одинаковы (даже названия типов в System.Data.OleDb и System.Data.SqlClient во многом совпадают). Главное различие между провайдерами заключается в том, что провайдер SQL не использует классические протоколы OLE DB или ADO и за счет этого обеспечивает значительный выигрыш в производительности.

В пространстве имен System.Data.Common определено множество абстрактных типов, которые обеспечивают общий интерфейс для всех управляемых провайдеров. Оба управляемых провайдера реализуют интерфейс IDbConnection, который

используется для конфигурирования и открытия сеанса подключения к источнику данных. Типы, которые реализуют другой интерфейс — IDbCommand, — используются для выполнения SQL-запросов к базам данных. IDataReader обеспечивает считывание данных при помощи однонаправленного курсора только для чтения. Типы, которые реализуют IDbDataAdapter, ответственны за заполнение объекта DataSet данными из базы данных.

В большинстве случаев нам не потребуется взаимодействовать с типами из пространства имен System.Data.Common напрямую. Однако для применения любого из управляемых провайдеров нам придется указать использование соответствующих пространств имен:

```
// Если мы будем использовать управляемый провайдер OLE DB
using System.Data;
using System.Data.OleDb;

// Если ны будем использовать управляемый провайдер SQL
using System.Data;
using System.Data.SqlClient;
```

Управляемый провайдер OLE DB

Работа со всеми управляемыми провайдерами очень похожа, и после того, как мы разберемся с одним управляемым провайдером, работа с другими не представит для нас больших трудностей. Мы начнем рассмотрение возможностей подключения к другим базам данных в ADO.NET с управляемого провайдера OLE DB. Этот провайдер потребуется нам во всех случаях, когда мы будем подключаться к источнику данных, отличному от MS SQL Server. Главные типы, используемые этим провайдером (они определены в пространстве имен System.Data.OleDb), представлены в табл. 13.12.

Таблица 13.12. Наиболее важные типы пространства имен System.Data.OleDb

Тип	Описание
OleDbCommand	Представляет запрос SQL, производимый к источнику данных
OleDbConnection	Представляет открытое соединение с источником данных
OleDbDataAdapter	Представляет соединение с базой данных и набор команд, используемых для заполнения объекта DataSet, а также обновления исходной базы данных после внесения изменений в DataSet
OleDbDataReader	Обеспечивает метод считывания потока данных из источника в одном направлении (вперед)
OleDbErrorCollection OleDbError OleDbException	OleDbErrorCollection представляет набор ошибок и предупреждений, возвращаемых источником данных. Сами эти ошибки и предупреждения представлены объектами OleDbError. При возникновении ошибки может быть сгенерировано исключение, представленное объектом OleDbException
OleDbParameterCollection OleDbParameter	Используются для передачи параметров процедуре, хранимой на источнике данных. Параметры представлены объектами OleDbParameter, весь набор — объектом OleDbParameterCollection

Установление соединения при помощи типа OleDbConnection

При работе с управляемым провайдером OLE DB первое, что мы должны сделать — установить соединение с источником данных при помощи типа OleDbConnection. Работа с OleDbConnection во многом напоминает работу с объектом Connection в классическом ADO. Для OleDbConnection предусмотрено использование строки подключения (connection string), состоящей из пар имя — значение. С ее помощью мы можем задать имя компьютера, к которому производится подключение, параметры безопасности подключения, имя базы данных, к которой производится подключение, а также, конечно, имя самого провайдера OLE DB. Полное описание всех возможных вариантов пар имя — значение можно найти в электронной документации к Visual Studio.NET, мы же ограничимся лишь наиболее важными моментами.

Создать строку подключения можно при помощи свойства OleDbConnection.ConnectionString (или передав ее в качестве параметра конструктора). Предположим, что нам необходимо подключиться к базе данных Cars на компьютере с именем BIGMANU через управляемый провайдер OLE DB. Код для этого может быть таким:

```
// Создаем строку соединения
OleDbConnection cn = new OleDbConnection();
cn.ConnectionString =
                                    "Provider=SQLOLEDB.1;" +
                                    "Integrated security=SSPI;" +
                                    "Persist Security Info=False;" +
                                    "Initial Catalog=Cars;" +
                                    "Data Source=BIGMANU;";
```

Data Source — это, конечно, имя компьютера, с которым мы устанавливаем соединение. Initial Catalog — имя базы данных, к которой мы подключаемся (в нашем случае Cars). Provider — это имя провайдера OLE DB, который будет использован для обращения к источнику данных. Возможные значения для Provider приведены в табл. 13.13.

Таблица 13.13. Наиболее часто используемые провайдеры OLE DB

Возможное значение для Provider в строке подключения	Описание
Microsoft.JET.OLEDB.4.0	Этот провайдер OLE DB используется для подключения к базам данных JET 9, то есть Access)
MSDAORA	Для подключения к базам данных Oracle
SQLOLEDB	Для подключения к базам данных MS SQL Server

После настройки строки подключения следующее, что мы должны сделать, — открыть сеанс соединения с источником данных. После этого мы выполняем нужные нам действия и разрываем соединение. Выглядеть все это может примерно так:

```
// Создаем строку соединения (при этом мы можем указать иня пользователя и пароль
// для подключения)
OleDbConnection cn = new OleDbConnection();
cn.ConnectionString =
                                    "Provider=SQLOLEDB.1;" +
```

```
                                        "Integrated security=SSPI;" +
                                        "Persist Security Info=False;" +
                                        "Initial Catalog=Cars;" +
                                        "Data Source=BIGMANU;";
cn.Open();
      // Выполняем различные операции
cn.Close();
```

Конечно, ConnectionString, OpenO и Close() — не единственные члены класса OleDbConnection. Кроме них, в этом классе предусмотрено множество членов, которые позволяют настраивать самые разные параметры подключения. Их краткий перечень представлен в табл. 13 14.

Таблица 13.14. Члены класса OleDbConnection

Член	Описание
BeginTransaction() CommitTransaction() RollbackTransaction()	Используются для того, чтобы программным образом начать транзакцию, завершить ее или отменить
Close()	Закрывает соединение с источником данных (наиболее рекомендуемый способ)
ConnectionString	Позволяет настроить строку подключения при установлении соединения или получить ее содержание
ConnectionTimeout	Позволяет получить или установить время тайм-аута при установке соединения
Database	Позволяет получить или установить название текущей базы данных во время подключения
DataSource	Позволяет получить или установить имя
Open()	Открывает соединение с базой данных, используя текущие настройки свойств соединения
Provider	Позволяет получить или установить имя провайдера
State	Позволяет получить информацию о текущем состоянии соединения

Построение команды SQL

Объектно-ориентированным представлением запроса на языке SQL в ADO.NET является класс OleDbCommand. Сам текст команды определяется через свойство OleDbCommand.CommandText. Множество типов ADO.NET принимают объект 01eDbCommand в качестве параметра для того, чтобы передать запрос к источнику данных. Помимо свойства CommandText, которое позволяет определить сам текст запроса, в классе 01eDbCommand предусмотрено также множество других членов, которые позволяют определить характеристики запроса (табл. 13.15).

Таблица 13.15. Члены класса OleDbCommand

Член	Описание
Cancel()	Прекращает выполнение команды
CommandText	Позволяет получить или задать текст запроса на языке SQL (возможно, с особенностями диалекта, определяемыми типом источника данных), который будет передан источнику данных

Член	Описание
CommandTimeout	Позволяет получить время тайм-аута при выполнении команды. По умолчанию это время равно 30 секундам
CommandType	Позволяет получить или установить значение, определяющее, как именно будет интерпретирован текст запроса, заданный через свойство. Позволяет получить ссылку на объект OleDbConnection, для которого используется данный объект CommandText
Connection	Позволяет получить ссылку на объект OleDbConnection, для которого используется данный объект OleDbCommand
ExecuteReader()	Возвращает объект OleDbDataReader
Parameters	Возвращает коллекцию параметров OleDbParameterCollection
Prepare()	Готовит команду к выполнению (например, она будет откомпилирована) на источнике данных

Работа с типом OleDbCommand производится очень просто, при этом, как и в случае с 01eDbConnection, в нашем распоряжении есть несколько способов добиться того же самого результата. В качестве примера приведем два варианта выполнения одной и той же команды SQL к объекту OleDbConnection. Для каждого из них будем считать, что у нас уже открыто соединение, представленное объектом OleDbConnection с именем cn:

```
// Первый вариант выполнения SQL-запроса
string strSQL1 = "Select Make from Inventory where Color='Red'";
OleDbCommand myCommand1 - new OleDbCommand(strSQL1, cn);

// Второй вариант выполнения SQL-запроса
string strSQL2 = "Select Make from Inventory where Color='Red'";
OleDbCommand myCommand2 = new OleDbCommand();
myCommand2.Connection - cn;
myCommand2.CommandText - strSQL2;
```

Работа с OleDbDataReader

После того как мы открыли соединение с источником данных и создали объект — команду SQL, следующая наша задача — передать эту команду (запрос) источнику данных. Это можно сделать несколькими способами, но использование OleDbDataReader — это наиболее простой, наиболее быстрый способ получения информации от источника данных... и наименее гибкий. Этот класс представляет однонаправленный (только вперед), доступный только для чтения поток данных, который за один раз возвращает одну строку в ответ на запрос SQL.

Класс OleDbDataReader очень полезен, когда нам необходимо последовательно обработать большое количество данных и в то же время не нужно выполнять с этими данными какие-либо операции в памяти, Например, если запрос возвращает 20 000 записей из таблицы для помещения их в текстовый файл, организовывать для них промежуточное хранение в оперативной памяти через объект DataSet было бы непрактичным (и с точки требований к оперативной памяти, и с точки зрения производительности). Гораздо лучше поток данных, возвращаемый из источника данных, перенаправить напрямую в другой поток, производящий запись в текстовый файл. В этой ситуации нам и потребуется объект OleDbDataReader. Обратите внимание, что в отличие от DataSet, при использовании OleDbDataReader со-

единение с базой данных не закрывается автоматически, а сохраняется активным до тех пор, пока мы не закроем его явным образом.

Проиллюстрируем все вышесказанное на примере. Предположим, что в нашем распоряжении есть класс, который производит простой запрос к нашей базе данных Cars при помощи метода OleDbCommand.ExecuteReader(). Этот метод возвратит объект класса OleDbDataReader, после чего мы можем воспользоваться методом OleDbDataReader.Read() для записи возвращаемых из базы данных записей в стандартный поток ввода-вывода:

```
public class OleDbDR
{
        static void Main(string[] args)
        {
                // Первый шаг; устанавливаем соединение
                OleDbConnection cn - new OleDbConnection();
                cn.ConnectionString - "Provider=SQLOLEDB.1;" +
                        "Integrated Security=SSPI;" +
                        "Persist Security Info=False;" +
                        "Initial Catalog=Cars;" +
                        "Data Source=BIGMANU;";
                cn.Open();

                // Второй шаг: создаем команду SQL
                string strSQL - "SELECT Make FROM Inventory WHERE Color='Red'";
                OleDbCommand myCommand = new OleDbCommand(strSQL, cn);

                // Третий шаг: получаем объект OleDbDataReader при помощи метода
                                                        ExecuteReader()
                OleDbDataReader myDataReader;
                myDataReader - myCommand.ExecuteReader();

                // Четвертый шаг: проходим циклон по всем возращаемым данным
                while (myDataReader.Read())
                {
                        Console.WriteLine("Red car: " + myDataReader["Make"].ToString());
                }

                myDataReader.Close();
                cn.Close();
        }
}
```

В результате будет выведен список всех автомобилей в базе данных Cars (рис. 13.25).

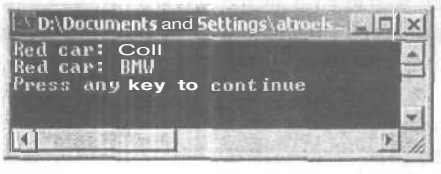


Рис. 13.25. Класс OleDbDataReader в действии

Поскольку объекты DataReader (вскоре мы познакомимся и с другими их разновидностями) обеспечивают лишь однонаправленный поток данных только для чтения, перемещаться по этим данным в приложении невозможно. Все, что мы

можем сделать, — считать каждую запись, возвращаемую запросом, и использовать ее в нашем приложении:

```
// Получаем значение из столбца 'Make'
Console.WriteLine("Red Car: {0}", myDataReader["Make"].ToString());
```

После того как DataReader станет нам больше не нужен, не забудем явным образом закрыть соединение с базой данных. Это делается при помощи метода OleDbDataReader.Close(). Конечно же, помимо методов Read() и Close(), в классе OleDbDataReader предусмотрены и другие методы, например для получения значения из указанного столбца в нужном нам формате (GetBoolean(), GetByte() и т. д.). Кроме того, можно отметить свойство FieldCount, которое возвращает количество столбцов в текущей записи.

Код приложения OleDbDataReader можно найти в подкаталоге Chapter 13.

Подключение к базе данных Access

Мы с вами уже умеем обращаться к базам данных SQL Server. Настало время узнать, как можно подключаться к другому распространенному виду баз данных — базам данных MS Access. Чтобы упростить себе задачу, мы просто изменим уже созданное только что нами приложение OleDbDataReader таким образом, чтобы считывать данные из базы данных Access (файла cars.mdb).

Как в классическом ADO, в ADO.NET, чтобы подключиться к другому источнику данных из уже готового кода, обычно достаточно лишь поменять содержимое строки подключения. Всего нам потребуется внести два изменения; *в* строке Provider поместить ссылку на JET вместо SQLOLEDB и в имени источника данных указать путь к файлу mdb:

```
// Изменяем источник данных в строке подключения:
OleDbConnection cn = new OleDbConnection();
cn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" + @"data source = D:\Chapter
13\Access DB\cars.mdb";
cn.Open();
```

После того как соединение с базой данных установлено, мы можем выполнять все необходимые операции точно так же, как и в предыдущем варианте нашего приложения. Единственное, о чем необходимо помнить: при подключении к базе данных Access можно использовать только типы из пространства имен System.Data.OleDb. Помните, что System.Data.SqlClient — только для баз данных SQL Server!

Выполнение хранимых процедур

Одно из важнейших решений, которые вам придется принять при создании распределенного приложения, — где будет реализована бизнес-логика. Один из возможных подходов — реализовать библиотеки кода с возможностью повторного использования (COM-компонент), которые будут управляться каким-либо вспомогательным процессом вроде Windows 2000 Component Services Manager. Другой подход — поместить бизнес-логику непосредственно на источник данных, реализовав ее в виде хранимых процедур (еще один распространенный подход — использовать одновременно и то и другое)

Хранимая процедура — это набор команд SQL, который в виде объекта со своим именем хранится в базе данных. Хранимая процедура может возвращать набор

строк (или просто значений указанных нами типов данных), а также принимать любое количество обязательных и необязательных параметров. По своему назначению хранимая процедура очень похожа на обычную функцию, например, в C#, с теми очевидными отличиями, которые вытекают из сущности хранимой процедуры как объекта базы данных.

Давайте добавим в нашу базу данных Cars простую хранимую процедуру GetPetName, которая будет принимать единственный параметр типа int (если мы создавали базу данных при помощи скрипта, то эта хранимая процедура должна быть уже создана). В качестве входящего параметра, конечно же, нами будет использован идентификатор автомобиля, а возвращаться будет прозвище автомобиля — значения столбца PetName (типа char). Синтаксис для создания хранимой процедуры может быть таким:

```
CREATE PROCEDURE GetPetName
       @carID int,
       @petName char(20) output
AS
SELECT @petName = PetName from Inventory where CarID = @carID
```

Теперь, когда хранимая процедура создана, нам необходимо каким-то образом ее вызвать. Первое, что, конечно, для этого нужно, — создать объект OleDbConnection, настроить строку подключения и открыть соединение с базой данных. После этого нам потребуется создать объект OleDbCommand, указав имя хранимой процедуры и установив нужное значение свойства CommandType:

```
// Открываем соединение с источником данных
OleDbConnection cn = new OleDbConnection();
cn.ConnectionString =
                                        "Provider=SQLOLEDB.1;" +
                                        "Integrated Security=SSPI;" +
                                        "Persist Security Info=False;" +
                                        "Initial Catalog=Cars;" +
                                        "Data Source=BIGMANU;";
cn.Open();

// А теперь создаем и настраиваем объект OleDbCommand для хранииой процедуры:
OleDbCommand myCommand = new OleDbCommand("GetPetName", cn);
myCommand.CommandType = CommandType.StoredProcedure;
```

Для свойства CommandType используются значения из одноименного перечисления. Значения этого перечисления приведены в табл. 13.16.

Таблица 13.16. Значения перечисления CommandType

Значение	Описание
StoredProcedure	Используется при настройке объекта OleDbCommand, который обеспечивает запуск на сервере баз данных хранимой процедуры
TableDirect	Для OleDbCommand достаточно будет указать имя таблицы — в результате будут возвращены все данные из этой таблицы
Text	Объект OleDbCommand будет представлять стандартную команду на языке SQL. Это значение используется по умолчанию

При выполнении обычных команд SQL о свойстве CommandType задумываться не надо, поскольку значение по умолчанию (Text) в этом случае вполне подходит,

Однако если команда представляет собой хранимую процедуру, то необходимо будет указать для свойства OleDbCommand.CommandType значение StoredProcedure.

Определяем параметры при помощи типа OleDbParameter

Обычно при запуске на выполнение хранимой процедуры нам необходимо определить передаваемые ей параметры. Для этой цели используется класс 01eDbParameter — объектная оболочка для параметров, которые передаются хранимой процедуре или возвращаются как результат ее работы. Для класса OleDbParameter предусмотрено большое количество свойств, которые позволяют определить имя, размер и тип данных для параметра, а также передается ли он хранимой процедуре или принимается от нее. Наиболее важные свойства типа OleDbParameter представлены в табл. 13.17.

Таблица 13.17. Свойства класса OleDbParameter

Свойство	Описание
DataType	Определяет тип параметра в терминах .NET
DbType	Позволяет получить или задать тип данных, используемый на источнике данных. Для этого свойства используются значения из перечисления OleDbType
Direction	Определяет, будет ли этот параметр передаваться на источник данных, возвращаться с источника данных или он сможет использоваться для передачи и в том и в другом направлении
IsNullable	Позволяет определить, будет ли данный параметр допускать пустые значения (значения типа NULL)
ParameterName	Позволяет получить или установить имя параметра
Precision	Позволяет получить или установить максимальное количество цифр, используемых для значения параметра (определяемое через свойство Value)
Scale	Позволяет получить или установить количество десятичных разрядов, используемых для значения параметра
Size	Позволяет получить или установить максимальный размер данных для данного параметра
Value	Позволяет получить или установить значение параметра

В нашей ситуации хранимая процедура принимает один параметр и один параметр возвращает. Соответствующий код приведен ниже. Обратите внимание, что параметры (то есть объекты OleDbParameter) добавляются в коллекцию ParametersCollection объекта 01eDbCommand при помощи свойства Parameters.

```
// Создаем объект для наших параметров
OleDbparameter theParam = new OleDbParameter();

// Параметр для передачи хранимой процедуре
theParam.ParameterName = "@carID";
theParam.DbType = OleDbType.Integer;
theParam.Direction = ParameterDirecton.Input;
// CarID = 1
theParam.Value = 1;
myCommand.Parameters.Add(theParam);

// Параметр для возврата значений из хранимой процедуры
theParam = new OleDbParameter();
```

```
theParam.ParameterName = "@PetName";
theParam.DbType = OleDbType.Char;
theParam.Size = 20;
theParam.Direction = ParameterDirection.Output;
myCommand.Parameters.Add(theParam);
```

Последнее, что осталось сделать — запустить программу на выполнение при помощи OleDbCommand.ExecuteNonQuery(). Обратите внимание, что для получения возвращаемых хранимой процедурой значений (в нашем случае — прозвища машины) используется свойство Value объекта OleDbParameter:

```
// Запускаем хранимую процедуру на выполнение
myCommand.ExecuteNonQuery;

// Выводим результат
Console.WriteLine("Stored Proc Info:");
Console.WriteLine("Car ID: " + myCommand.Parameters["@carID"].Value;
Console.WriteLine("PetName: " + myCommand.Parameters["@petName"].Value);
```

Результат работы программы представлен на рис. 13.26.

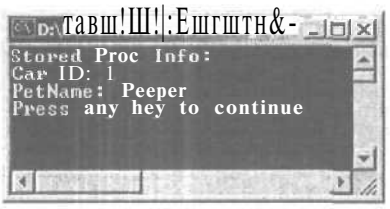


Рис. 13.26. Запуск хранимой процедуры

Код приложения OleDbStoredProc можно найти в подкаталоге Chapter 13.

Тип OleDbDataAdapter

К этому моменту мы уже узнали, как подключиться к источнику данных при помощи класса OleDbConnection, выполнить команду SQL, используя типы OleDbCommand и OleDbParameter, и получить однонаправленный поток данных при помощи OleDbDataReader. Всего этого вполне достаточно для выполнения хранимой процедуры или перенаправления потока данных из базы данных в поток вывода. Однако очень часто требуется заполнить полученными с сервера данными объект DataSet и выполнить с ними определенные операции. Наиболее гибкий способ, который позволяет это сделать, — использование класса OleDbDataAdapter.

Основное назначение этого класса — извлечь информацию из источника данных и заполнить ею объект DataTable в DataSet при помощи метода OleDbDataAdapter.Fill(). Метод Fill() многократно перегружен, вот два наиболее часто используемых варианта (возвращаемое значение int позволяет получить информацию о количестве записей, полученных из источника данных):

```
// Заполняем объект DataSet данными, полученными из таблицы на источнике данных
// с указанным именем:
public int Fill(DataSet yourDS, string tableName);

// Снова заполняем данными, но только теми, которые находятся в указанных ими
// границах
public int Fill(DataSet yourDS, string tableName, int startRecord, int maxRecord);
```

Конечно, перед тем как вызывать этот метод, нам потребуется уже созданный объект OleDbDataAdapter. Конструктор OleDbDataAdapter также многократно перегружен, но обычно необходимо указать информацию о параметрах подключения к базе данных и команду SELECT на языке SQL, которая будет использована для заполнения DataTable.

OleDbDataAdapter позволяет не только заполнять объект DataTable внутри DataSet данными, полученными из источника, но и помещать измененные данные обратно в источник данных при помощи стандартных команд SQL. В табл. 13.18 представлены члены класса OleDbDataAdapter, которые позволяют это сделать, а также некоторые другие важнейшие члены этого класса.

Таблица 13.18. Наиболее важные члены OleDbDataAdapter

Член	Описание
DeleteCommand InsertCommand SelectCommand UpdateCommand	Используются для определения того, какая именно команда SQL будет передана на источник данных при вызове метода Update(). Каждое из этих свойств определяется при помощи объектов OleDbCommand
Fill()	Заполняет указанную таблицу в DataSet определенным пользователем количеством записей
GetFillParameters()	Возвращает все параметры, использованные при выполнении запроса SELECT к источнику данных
Update()	Вызывает соответствующие команды INSERT, **UPDATE, DELETE** к источнику данных для каждой вставленной, измененной или удаленной строки в таблице объекта DataSet

При использовании свойств DeleteCommand, InsertCommand, UpdateCommand, SelectCommand объект OleDbDataAdapter автоматически переводит внесенные нами изменения в таблицу данных в DataSet в соответствующие команды на языке SQL, сохраняя, таким образом, внесенные нами в DataSet изменения в источнике данных. Эти свойства позволяют определить соответствующие команды SQL в деталях. Однако прежде чем мы приступим к их рассмотрению, мы познакомимся с тем, как можно использовать OleDbDataAdapter в коде программы для заполнения данными объекта DataSet.

Заполнение данными объекта DataSet при помощи OleDbDataAdapter

Заполнить данными объект DataSet (в котором имеется только одна таблица) при помощи OleDbDataAdapter можно, если использовать следующий код:

```
public class MyOleDbDataAdapter
{
        // Шаг 1: открываем соединение с базой данных Cars
        OleDbConnection cn = new OleDbConnection();
        cn.ConnectionString =
                                        "Provider=SQLOLEDB.1;" +
                                        "Integrated security=SSPI;" +
                                        "Persist Security Info=False;" +
                                        "Initial Catalog=Cars;" +
```

```
                                        "Data Source=BIGMANU;";
        cn.Open();

        // Шаг 2: Создаем OleDbDataAdapter при помощи команды SELECT
        string sqlSELECT = "SELECT * FROM Inventory";
        OleDbDataAdapter dAdapt = new 01eDbDataAdapter(sqlSELECT, cn);

        // Шаг 3: Создаем и заполняем объект OataSet, а потом закрываем соединение
        DataSet myDS = new DataSet("CarsDataSet");
        try
        {
              dAdapt.Fill(myDS, "Inventory");
        }
        catch(Exception ex)
        {
              Console.WriteLine(ex.Message);
        }
        finally
        {
              cn.Close();
        }

        // Вспомогательная функция для вывода содержимого таблицы
        PrintTable(myDS);
        return 0;
}
```

Обратите внимание, что создание таблицы Inventory (объекта DataTable) было совсем непохоже на то, как мы делали это в первой части главы. Вместо того чтобы создавать OataTable из отдельных элементов, а затем добавлять ее в OataSet, мы просто указали имя создаваемой таблицы (Inventory) в качестве второго параметра для метода Fi ll(). Все остальное этот метод сделал автоматически: создал объект DataTable, присвоил ему указанное нами имя, создал внутри DataTable объекты DataColumn и заполнил таблицу строками из источника данных, полученных в результате выполнения запроса SELECT. Команда SELECT была передана источнику данных как параметр конструктора OleDbDataAdapter:

```
// Команда SELECT была определена просто как переиенная типа string
string sqlSELECT - "SELECT * FROM Inventory";
OleDbDataAdapter dAdapt = new OleDbDataAdapter(sqlSELECT, cn);
```

С точки зрения объектно-ориентированного программирования, конечно, привычнее было бы создать для команды SELECT отдельный объект (вместо переменной string) и использовать именно его. Можно сделать и так. Понятно, что команду SELECT в этом случае будет представлять объект 01 eDbCommand. Чтобы связать этот объект с объектом OleDbDataAdapter, используется свойство SelectCommand:

```
// Создаем объект OleDbCommand, представляющий команду SELECT
OleDbCommand selectCmd = new OleDbCommand("SELECT * FROM Inventory", cn);

// Создаем объект OleDbDataAdapter и привязываем к нему объект OleDbCommand
OleDbDataAdapter dAdapt = new OleDbDataAdapter();
dAdapt.SelectCommand = selectCmd;
```

В любом случае результат будет одним и тем же (рис. 13.27).

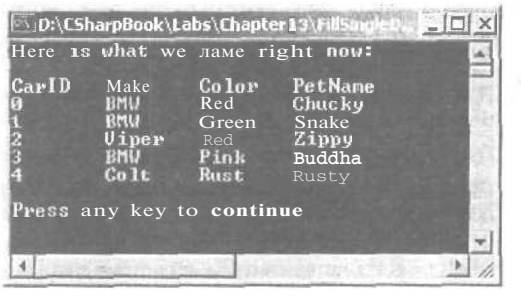


Рис. 13.27. Объект OleDbDataAdapter в действии

Однако, чтобы программа действительно заработала, нам осталось создать ее последнюю часть — метод PrintTable(). Он не слишком сложен:

```
public static void PrintTable(DataSet ds)
{
        // Получаем таблицу Inventory из объекта DataSet
        ConsoleWriteLine("Here is what we have right now: \n");
        DataTable invTable = ds.Tables["Inventory"];

        // Выводим имена столбцов
        forCint curCol = 0: curCol < Invtable.Columns.Count; curCol++)
    {
    Console.WriteLine(invTable.Columns[curCol].ColumnName.Trim() + "\t");
        }
        Console.WriteLine();

        // Выводим значения из каждого поля таблицы
        for(int curRow = 0; curRow < invTable.Rows.Count; curRow++)
        {
                forCint curCol = 0: curCol < Invtable.Columns.Count; curCol++)
                {
                        Console.Write(InvTable.Rows[curRow][curCol].ToString().Trim() +
                                                                                "\t");
                }
                Console.WriteLine()
        }
}
```

Код приложения FillSingleDSWithAdapter можно найти в подкаталоге Chapter 13.

Работа с управляемым провайдером SQL

Перед тем как мы приступим к вставке, изменению и удалению записей в источнике данных при помощи объектов DataAdapter, мы рассмотрим еще одну тему — особенности работы с управляемым провайдером SQL. Как уже говорилось, по своим функциональным возможностям этот провайдер схож с управляемым провайдером OLE DB, но он предназначен только для работы с базами данных на сервере MS SQL Server и хорошо оптимизирован именно для такой работы.

Типы, которые составляют управляемый провайдер SQL, определены в пространстве имен System.Data.SqlClient. Список основных типов приведен в табл. 13.19. В нем мы найдем много общего с уже знакомыми нам типами из пространства имен System.Data.OleDb.

Таблица 13.19. Наиболее важные типы пространства имен System.Data.SqlClent

Тип	Описание
SqlCommand	Представляет запрос SQL, производимый к источнику данных — SQL Server
SqlConnection	Представляет открытое соединение с источником данных
SqlDataAdapter	Представляет соединение с базой данных и набор команд, используемых для заполнения объекта DataSet, а также обновления исходной базы данных после внесения изменений в DataSet
SqlDataReader	Обеспечивает метод считывания потока данных из источника в одном направлении (вперед)
SqlErrors SqlError SqlException	SqlErrors представляет набор ошибок и предупреждений, возвращаемых источником данных. Сами эти ошибки и предупреждения представлены объектами SqlError. При возникновении ошибки может быть сгенерировано исключение, представленное объектом SqlException
SqlParameterCollection SqlParameter	Используются для передачи параметров хранимой процедуре на источнике данных. Параметры представлены объектами SclParameter

Работа с этими типами данных практически идентична работе с аналогичными типами данных *из* пространства имен System.Data.01eDb. Однако, чтобы сделать работу с типами данных управляемого провайдера SQL более привычной, в оставшейся части этой главы мы будем иметь дело именно с ними.

Пространство имен System.Data.SqlTypes

Этот раздел также можно рассматривать как «заметки на полях». При использовании управляемого провайдера SQL очень удобно использовать классы, которые предназначены для представления «родных» типов данных SQL Server 7.0 и SQL Server 2000. Эти классы определены в пространстве имен System. Data. Sql Types. Их перечень приведен в табл. 13.20.

Таблица 13.20. Типы пространства имен System.Data.SqlTypes

Тип	Типы данных — аналогов на SQL Server
SqlBinary	binary, varbinary, timestamp, image
SqlInt64	bigint
SqlBit	**bit**
SqlDateTime	datetime, smalldatetime
SqlNumeric	decimal
SqlDouble	float
SqlInt32	int
SqlMoney	money, smallmoney
SqlString	nchar, ntext, nvarchar, sysname, text, varchar, char
SqlNumeric	numeric
SqtSingle	real
SqlInt16	smallint
System.Object	sql_variant
SqlByte	tinyint
SqlGuid	uniqueidentifier

Вставка новых записей при помощи SqlDataAdapter

Мы сделали крутой поворот от типов управляемого провайдера OLE DB к типам управляемого провайдера SQL, но наша задача осталась прежней: выяснить, как обеспечить запись измененных данных в базу данных на источнике при помощи объектов DataAdapter. Первое, с чем мы познакомимся — со вставкой новых записей при помощи SqlDataAdapter (еще раз отметим, что с точки зрения использования в коде программы разницы между SqlDataAdapter и OleDbDataAdapter почти никакой нет). Как обычно, первое, что необходимо сделать — открыть соединение с базой данных:

```
public class MySqlDataAdapter
{
        public static void Main()
        {
                // Шаг 1: Создаем соединение и объект SqlDataAdapter
                // (с командой SELECT)
                SqlConnection cn = new
                                SqlConnection("server=(local);uid=sa;pwd=;database=Cars");

                SqlDataAdapter dAdapt = new SqlDataAdapter("Select * from Inventory", cn);

                // Шаг 2: Если в таблице уже присутствует запись с номером, совпадающий
                с номером вставляемой нами записи, удаляем ее
                cn.Open();
                SqlCommand killCmd = new SqlCommand("Delete from Inventory where CarID =
                                                                        '1111'", cn);
                killCmd.ExecuteNonQuery();
                cn.Close();
        }
}
```

Первое, что бросается в глаза, — изменилось содержание строки подключения. При работе с управляемым провайдером SQL не указывается значение для Provider (поскольку мы всегда подключаемся к SQL Server), кроме того, в целом используется несколько другой набор пар имя — значение. После того как соединение с использованием строки подключения установлено, мы уже действуем привычным путем: создаем объект SqlDataAdapter и определяем для него текст запроса SQL через конструктор.

Шаг второй нашего кода — это устранение возможных проблем, которые могут быть связаны с тем, что в таблице в базе данных уже есть запись с совпадающим значением CarID. В этой ситуации нам просто не дадут произвести вставку (помните, что на столбец CarID наложено ограничение первичного ключа?). Поэтому мы на всякий случай решаем эту проблему наиболее простым способом.

После того как соединение установлено, следующая наша задача — создать новый объект для представления команды SQL INSERT. Конечно же, для представления этой команды будет использован объект SqlCommand, а для параметров команды INSERT — объект SqlParameter:

```
public static void Main()
{
        ...
        // Шаг 3: создаем команду INSERT
        dAdapt.InsertCommand = new SqlCommand("INSERT INTO Inventory" +
                                        "(CarID, Make, Color, PetName) VALUES" +
                                        "(@CarID, @Make, @Color, @PetName)", cn)";
```

```
        // Шаг 4: Начинаем работу по созданию параметров для каждого столбца
        // в таблице Inventory
        SqlParameter workParam = null;

        // Параметр для столбца CarID
        workParam = d.Adapt.InsertCommand.Parameters.Add(new SqlParameter("@CarID",
                                                         SqlDbType.Int));
        workParam.SourceColumn = "CarID";
        workParam.SourceVersion = DataRowVersion.Current;

        // Параметр для столбца Make
        workParam = d.Adapt.InsertCommand.Parameters.Add(new SqlParameter("@Make",
                                                         SqlDbType.VarChar));
        workParam.SourceColumn = "Make";
        workParam.SourceVersion = DataRowVersion.Current;

        // Параметр для столбца Color
        workParam = d.Adapt.InsertCommand.Parameters.Add(new SqlParameter("@Color",
                                                         SqlDbType.VarChar));
        workParam.SourceColumn = "Color";
        workParam.SourceVersion = DataRowVersion.Current;

        // Параметр для столбца PetName
        workParam = d.Adapt.InsertCommand.Parameters.Add(new SqlParameter("@PetName",
                                                         SqlDbType.VarChar));
        workParam.SourceColumn = "PetName";
        workParam.SourceVersion = DataRowVersion.Current;
}
```

Теперь мы готовы к тому, чтобы произвести вставку новой строки в DataSet (предварительно мы его создадим и заполним данными из источника) и записать произведенные изменения из DataSet обратно в источник данных. Кроме того, для наглядности мы также выведем то, что у нас получилось, на консоль при помощи ранее созданной нами функции PrintTable():

```
public static void Main()
{
        ...
        // Шаг 5: Создаем объект DataSet и заполняем его данными из базы данных
        // на источнике:
        DataSet myDS = new DataSet();
        dAdapt.Fill(myDS, "Inventory");
        PrintTable(myDS);

        // Шаг 6: добавляем новую запись в таблицу в DataSet
        DataRow newRow = myDS.Tables["Inventory"].NewRow();
        newRow["CarID"] = 1111;
        newRow["Make"] = "SlugBug";
        newRow["Color"] = "Pink";
        newRow["PetName"] = "Cranky";
        myDS.Tables["Inventory"].Rows.Add(newRow);

        // Шаг 7: Передаем изменения на источник данных и проверяем это
        try
        {
                dAdapt.Update(myDS, "Inventory");
                myDS.Dispose ();
                myDS = new DataSet();
                dAdapt.Fill(myDS, "Inventory");
```

```
            PrintTable(myDS);
        }
        catch(Exception e)
        {
            Console.Write(e.ToString());
        }
    }
```

Результат работы нашей программы представлен на рис. 13.28.

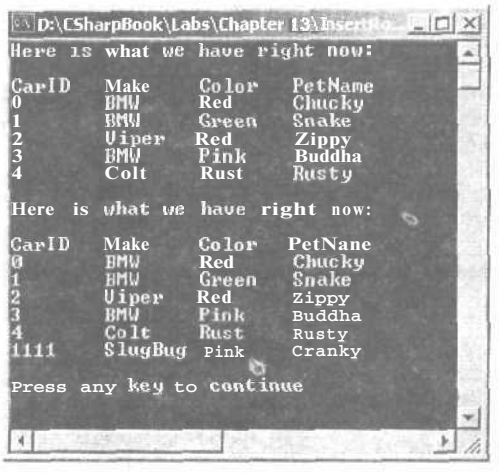


Рис. 13.28. Применение свойства InsertCommand

Код приложения InsertRowsWithSqlAdapter можно найти в подкаталоге Chapter 13.

Изменение записей в таблице при помощи SqlDataAdapter

Процесс изменения существующих строк в таблице на источнике данных очень похож на процесс вставки новых строк. Как обычно, мы начинаем с создания объекта SqlDataAdapter и открытия соединения с базой данных. Далее наша задача — воспользоваться свойством InsertCommand:

```
public static void Main()
{
    // Шаг 1: Создаем объект SqlDataAdapter и открываем соединение (аналогично
    // предыдущему примеру, поэтому эту часть кода опускаем)
    ...

    // Шаг 2: Создаем команду UPADTE
    dAdapt.UpdateCommand = new SqlCommand("UPDATE Inventory SET Make=@Make,
                        Color=@Color, PetName=@PetName WHERE CarID = @CarID", cn);

    // Шаг 3: Создаем объекты параметров для каждого столбца в таблице Inventory.
    // Все так же, как и в предыдущем примере, только мы помещаем эти параметры
    // в коллекцию ParameterCollection для UpdateCommand. Например:
    SqlParameter workParam = null;
```

```
        workParam = dAdapt.UpdateCommand.Parameters.Add(new SqlParameter("@CarID",
                                                                SqlDbType.Int));
        workParam.SourceColumn = "CarID";
        workParam.SourceVersion = DataRowVersion.Current;

        // Делаем то же самое для столбцов PetName, Make и Color
        ...

        // Шаг 4; Заполняем данными объект DataSet
        DataSet myDS - new DataSet();
        dAdapt.Fill(myDS, "Inventory");
        PrintTable(myDS);

        // Шаг 5: Меняем значения столбцов во второй строке таблицы на 'FooFoo'
        DataRow changeRow - myDS.Tables["Inventory"].Row[1];
        changeRow["Make"] = "FooFoo";
        changeRow["Color"] = "FooFoo";
        changeRow["PetName"] = "FooFoo";


        // Шаг 6: Сохраняем данные в базе данных и выводим значения на консоль
        try
        {
              dAdapt.Update(myDS, "Inventory");
              myDS.Dispose();
              myDS = new DataSet();
              dAdapt.Fill(myDS, "Inventory");
              PrintTable(myDS);
        }
        catch(Exception e)
        { Console.Write(e.ToString()); }
}
```

Результат работы программы представлен на рис. 13.29.

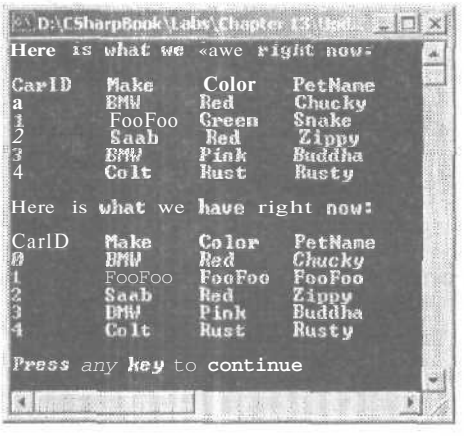


Рис. 13.29. Внесение изменений в существующие записи в базе данных

Код приложения UpdateRowsWitSqlAdapter можно найти в подкаталоге Chapter 13.

Автоматическое создание команд SQL

Мы только что научились вставлять, удалять и изменять данные **при помощи объектов** 01 eDbDataAdapter и Sql DataAdapter. **Работу с** этими **типами не назовешь особенно сложной, но создание** отдельного **параметра для каждого из** столбцов таблицы **и передача их через свойства** InsertCommand, UpdateCommand и DeleteCommand **может показаться несколько утомительной.** Наверно, **вы уже догадываетесь, что есть и более удобный способ.**

Этот способ заключается в применении класса SqlCommandBuilder. Если мы работаем с объектом DataTable, состоящим из единственной таблицы (а не результатом объединения нескольких таблиц), то Sql CommandBuilder автоматически настроит свойства InsertCommand, UpdateCommand и DeleteCommand в соответствии с тем, что мы изначально использовали для свойства SelectCommand! Однако применяться SqlCommandBuilder может не всегда. Помимо ограничений, связанных с объединениями, о которых мы уже говорили, должно обязательно выполняться еще одно условие: для таблицы должен быть определен первичный ключ, и этот ключ должен использоваться в исходном запросе — команде SELECT. Главное же преимущество применения SqlCommandBuilder заключается в том, что нам нет необходимости создавать объекты Sql Parameter вручную.

Рассмотрим применение SqlCommandBuilder на примере. Предположим, что в нашем распоряжении имеется форма Windows, на которой расположен элемент управления DataGrid. Пользователь может редактировать записи прямо в DataGrid, а по завершении он должен нажать специальную кнопку, чтобы сохранить изменения в базе данных. Конструктор для нашей формы будет выглядеть так:

```
public class mainForm : System.Windows.Forms.Form
{
        private SqlConnection cn = new
                        SqlConnection("server=(local);uid=sa;pwd=;database=Cars");

        private SqlDataAdapter dAdapt;

        privateSqlCommandBuilderinvBuilder;
        private DataSet myDS = new DataSet();

        private System.Windows.Forms.DataGrid dataGrid1;
        private System.Windows.Forms.Button btnUpdateData;
        private System.ComponentModel.Container components;

        public mainForm();
        f
                InitializeComponents();

                // Создаем исходную команду SELECT
                dAdapt = new SqlDataAdapter("SELECT * from Inventory", cn);

                // А сейчас команды INSERT, UPDATE и DELETE будут сгенерированы
                // автоматически
                invBuider = new SqlCommandBuilder(dAdapt);

                // Заполняем DataSet и привязываем к нему DataGrid
                dAdapt.Fill(myDS, "Inventory");
```

```
            dataGrid1.DataSource = myDS.Tables["Inventory"].DefaultView;
        }
...
!
```

Теперь у SqlDataAdapter есть вся необходимая информация для записи данных из DataGrid обратно в источник данных. Код для события Click нашей кнопки будет таким:

```
private void btnUpdateData_Click(object sender, System.EventArgs e)
{
        try
        {
            DataGrid1.Refresh();
            dAdapt.Update(myDS, "Inventiry");
        }
        catch(Exception ex)
        {
            MessageBox.Show(ex.ToString());
        }
}
```

Как обычно, мы вызвали метод Update() и указали объект DataSet и таблицу в нем, в которую следует записать изменения. Наше приложение будет выглядеть примерно так, как представлено на рис. 13.30 (перед нажатием на кнопку убедимся, что мы вышли из режима редактирования!).

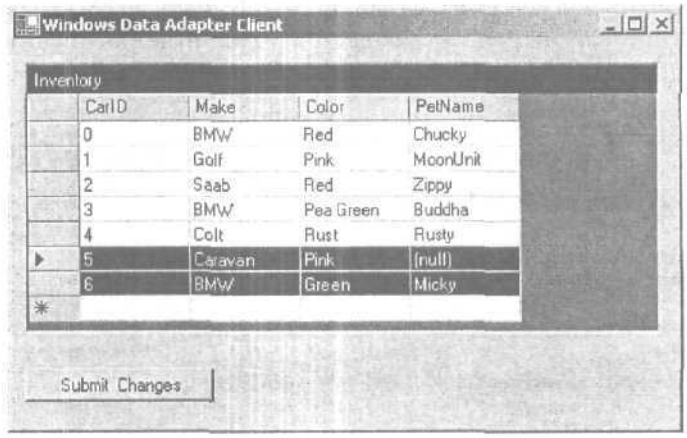


Рис. 13.30. Наше приложение, использующее SqlCommandBuilder

Я думаю, вы согласитесь со мной, что применение автоматически генерируемых команд SQL гораздо проще, чем создание таких команд вручную. Однако, конечно же, все имеет свои недостатки. К примеру, если объект DataTable создан на основе объединения двух таблиц в источнике данных, использовать эту технологию нам не удастся. Кроме того, создание команд вручную предоставляет в наше распоряжение гораздо больше возможностей в различных специфических ситуациях.

Код приложения WinFormSqlAdapter можно найти в подкаталоге Chapter 13.

Заполнение объекта DataSet с несколькими таблицами и добавление объектов DataRelations

Мы завершим эту главу тем, что создадим приложение, очень похожее на то, что было создано в самом начале этой главы. Его интерфейс нам уже знаком (рис. 13.31): на форме три элемента управления DataGrid, в качестве источников данных для которых служат записи, полученные из таблиц Inventory, Orders и Customers базы данных Cars. Единственная кнопка формы предназначена для передачи измененных данных из DataGrid обратно в базу данных.

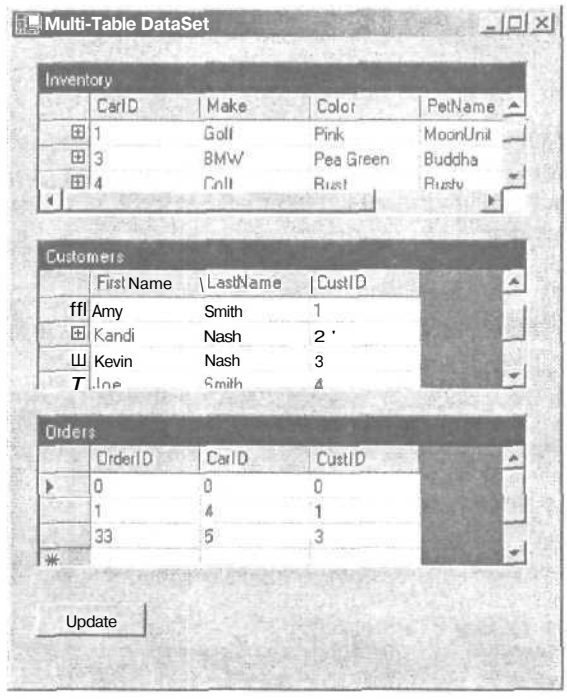


Рис. 13.31. Информация из многотабличного объекта DataSet выведена на форму

Чтобы сделать все максимально простым, мы будем использовать для каждого из трех объектов SqlDataAdapter (по одному на каждую таблицу) автоматически генерируемые команды SQL. Прежде всего подготовим все объекты, которые понадобятся нам при использовании нашей формы:

```
public class mainForm : System.Windows.Forms.Form
{
        private System.Windows.Forms.DataGrid custGrid;
        private System.Windows.Forms.DataGrid inventoryGrid;
        private System.Windows.Forms.Button btnUpdate;
        private System.Windows.Forms.DataGrid OrdersGrid;
        private System.ComponentModel.Container components;

        // Соединение с базой данных
```

```
    private SqlConection cn = new
                SqlConnection("server=(Local);uid=sa;pwd=;database=Cars");

    // Объекты DataAdapter (для каждой таблицы)
    private SqlDataAdapter invTableAdapter;
    private  SqlDataAdapter custTableAdapter;
    private SqlDataAdapter ordersTableAdapter;

    // Объекты CommandBuilder (для каждой таблицы)
    private SqlCommandBuilder invBuilder = new SqlCommandBuilder;
    private SqlCommandBuilder orderBuilder = new SqlCommandBuilder;
    private SqlCommandBuilder custBuilder = new SqlCommandBuilder;

    // Сам объект DataSet
    DataSet carsDS = new OataSetC);
...
}
```

Вся работа по созданию переменных для работы с данными и заполнению объекта DataSet производится в конструкторе формы. В нем также будет производиться вызов к вспомогательной функции BuildTableRelationShip(). Код для конструктора будет выглядеть так:

```
public mainForm()
{
    InitializeComponent();

    // Создаем объекты DataAdapter
    InvTableAdapter = new SqlDataAdapter("SELECT * from Inventory", cn);
    custTableAdapter = new SqlDataAdapter("SELECT * from Customers", cn);
    ordersTableAdapter = new SqlDataAdapter("SELECT * from Orders", cn);

    // Применяем автоматическую генерацию команд SQL
    invBuilder = new SqlCommandBuilder(invTableAdapter);
    orderBuilder = new SqlCommandBuilder(ordersTableAdapter);
    custBuilder = new SqlCommandBuilder(custTableAdapter);

    // Заполняем таблицы в DataSet
    invTableAdapter.Fill(carsDS, "Inventory");
    custTableAdapter.Fill(carsDS, "Customers");
    orderTableAdapter.Fill(carsDS, "Orders");

    // Создаем отношения между таблицами
    Bui ldTableRelationshi p();
}
```

Вспомогательная функция Bui l dTabl eRel ationship() делает именно то, о чем вы, наверное, уже догадались по ее названию — создает отношения между таблицами. Код для нее будет выглядеть в полном соответствии с тем, с чем мы уже сталкивались в этой главе:

```
private void BuildTableRelationship()
{
    // Создаем объект DataRelation
    DataRelation dr = new DataRelation("CustomerOrder",
                carsDS.Tables["Customers"].Columns["CustID"],
                carsDS.Tables["Orders"].Columns["CustID"]);
```

```
        // Добавляем отношение в DataSet
        carsDS.Relations.Add(dr);

        // Создаем еще объект DataRelation
        dr = new DataRelation("InventoryOrder",
                    carsDS.Tables["Inventory"].Columns["CarID"],
                    carsDS.Tables["Orders"].Columns["CarID"]);

        // Добавляем отношение в DataSet
        carsDS.Relations.Add(dr);

        // Заполняем элементы управления DataGrid
        inventoryGrid.SetDataBinding(carsDS, "Inventory");
        custGrid.SetDataBinding(carsDS, "Customers");
        OrdersGrid.SetDataBinding(carsDS, "Orders");
}
```

Теперь, когда объект DataSet заполнен, мы можем работать с ним без какого-либо соединения с базой данных — полностью локально. Просто произведем все необходимые операции данные прямо в элементах управления DataGrid. После того как все необходимые изменения внесены, нажмем кнопку Update. Код для события Click этой кнопки у нас будет таким:

```
private void btnUpdate_Click(object sender. SystemEventargs e)
{
        try
        {
                invTableAdapter.Update(carsDS, "Inventory");
                custTableAdapter.Update(carsDS, "Customers");
                ordersTableAdapter.Update(carsDS, "Orders");
        }
        catch(Exception ex)
        {
                MessageBox.Show(ex.Message);
        }
}
```

К этому моменту мы уже научились работать с управляемыми провайдерами OLE DB и SQL и узнали, как производится работа с DataSet и связанными с ним типами. Однако, конечно, в ADO.NET остается еще немало интересных вещей, таких как работа с транзакциями, вопросы безопасности при подключении к базе данных и т, п. К сожалению, в рамках этой книги мы не можем рассмотреть эти темы.

Единственный момент, о котором хотелось бы еще упомянуть, — это то, что в Visual Studio.NET помещены специальные мастера для работы с подключениями к источникам данных, которые во многих ситуациях позволят сэкономить множество времени. Например, если мы перетащим элемент управления Data с панели Toolbox на нашу форму, мы сможем воспользоваться мастерами для создания строк подключения для типов SqlConnection и OleDbConnection, команд SELECT, INSERT, DELETE и UPDATE и многих других вещей. После того как в этой главе мы все проделали вручную, разобраться с тем, что создают эти мастера, вам не составит труда.

Код приложения MultiTableDataSet можно найти в подкаталоге Chapter 13.

Подведение итогов

ADO.NET — это новая технология доступа к данным, специально разработанная для применения в многоуровневых приложениях, в которых обеспечить постоянное соединение с источником данных не представляется возможным. Большинство типов, которые необходимы для обеспечения взаимодействия со строками, столбцами, таблицами и представлениями, находятся в пространстве имен System.Data. В пространствах имен System.Data.SqlClient и System.DataOleDb определены типы, которые позволяют устанавливать соединение с источниками данных MS SQL Server и OLE DB.

Главный тип в ADO.NET — это класс DataSet. DataSet предназначен для представления в оперативной памяти любого количества таблиц, отношений между ними, ограничений и выражений. То, что при помощи DataSet на клиенте представлены не только таблицы, но и отношения между ними, позволяет производить переходы между таблицами без необходимости всякий раз устанавливать соединение с удаленным источником данных.

Типы OleDbDataAdapter и SqlDataAdapter позволяют (при помощи свойств SelectCommand, InsertCommand, UpdateCommand и DeleteCommand) вносить изменения в исходную базу данных на источнике данных. В ADO.NET предусмотрено еще множество замечательных возможностей, но и с теми, которые были рассмотрены в данной главе, мы уже можем создавать вполне работоспособные приложения, обращающиеся к базам данных.

Разработка web-приложений и ASP.NET 14

До настоящего момента все приложения, которые мы разрабатывали, были консольными приложениями или приложениями Windows Forms. В этой главе мы познакомимся с новым типом приложений — web-приложениями, для доступа к которым клиентам нужен лишь браузер. В начале мы рассмотрим главные «атомы Web», без которых не обходится ни одно web-приложение — HTML, запросы HTTP (POST и GET), применение скриптов, выполняемых в браузере клиента (JavaScript), а также классические ASP. Конечно, если вы уже знакомы с этими темами, вы вполне можете пропустить эту часть.

Затем мы рассмотрим вопросы, связанные с применением ASP.NET. Как мы увидим, ASP.NET предлагает гораздо боле надежную модель создания web-приложений, нежели классические ASP. Например, мы можем разделить логику представления на HTML и бизнес-логику при помощи техники, называемой Codebehind. Кроме того, при создании web-приложений на ASP.NET мы можем использовать «настоящие» языки программирования, такие как C# и VB.NET, а не только интерпретируемые языки скриптов. Мы познакомимся с архитектурой web-приложения, с важнейшим типом Page и со свойствами, пришедшими из классического ASP, такими как Request, Response, Session и Application.

В самом конце главы мы рассмотрим серверные элементы управления (WebForm Controls) и события сервера. Одна из главных задач этой главы — подготовиться к созданию web-служб ASP.NET, о чем пойдет речь в следующей главе.

Web-приложения и web-серверы

Перед тем как погружаться в среду ASP.NET, мы должны рассмотреть основы архитектуры web-приложений и некоторые базовые web-технологии. Вначале дадим не очень строгое определение: web-приложение — это набор взаимосвязанных файлов (*.htm, *.asp, *,aspx, файлов изображений и т. п.), а также связанных с ними компонентов (двоичных файлов .NET или классического COM), которые размещены на web-сервере.

Web-сервер — это программный продукт, на котором размещаются ваши web-приложения и который обычно обеспечивает набор связанных с web-приложениями служб, таких как интегрированные средства обеспечения безопасности, поддержка протокола FTP, поддержка средств передачи электронной почты и т. п. Web-сервер уровня предприятия от Microsoft называется Internet Information Server (IIS). На момент написания этой книги последней версией IIS была версия 5.0, которая поставлялась с Windows 2000 как часть операционной системы.

При создании web-приложений с использованием классических ASP или ASP.NET нам обязательно придется — прямо или опосредованно — работать с IIS. Однако, если мы работаем под Windows 2000 Professional, будем помнить, что по умолчанию в этой операционной системе IIS не устанавливается. На всем протяжении этой главы IIS нам будет необходим, и поэтому, если он еще у нас не установлен, лучше сделать это прямо сейчас. Это очень просто: в Панели управления выберем Add/Remove Programs (Добавить/Удалить программы) и найдем пункт Add/Remove Windows Components (Добавить/Удалить компоненты Windows).

После того как установка IIS будет завершена, проще всего управлять им из консоли MMC, которая называется Internet Services Manager (ее можно найти в Administrative Tools). В этой главе мы не будем использовать виртуальные web-серверы и ограничимся использованием лишь web-сервера по умолчанию. Он помечен в окне Internet Services Manager как Default Web Site (рис. 14.1).

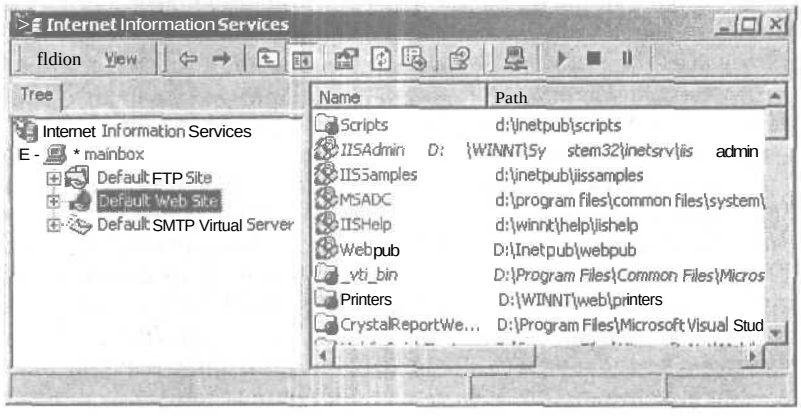


Рис. 14.1. Администрирование IIS

Что такое виртуальные каталоги

На одном сервере LIS может находиться множество web-приложений. Каждое из этих web-приложений должно размещаться в своем *виртуальном каталоге* (virtual directory). Виртуальному каталогу на web-сервере соответствует физический каталог на диске. Предположим, что мы создали web-приложение FrogsAreUs. Из внешнего мира к нему можно будет обратиться по адресу URL http://www.FrogsAreUs.com (если мы зарегистрировали это доменное имя в системе DNS), а на нашем компьютере этому приложению будет соответствовать физический каталог, например C:\FrogSite. И именно в этом физическом каталоге будут находиться файлы, из которых состоит наше web-приложение.

Давайте создадим простое web-приложение, которое будет называться Cars. Первое, что нам потребуется сделать, — создать на компьютере новый каталог, в котором будут храниться файлы нашего web-приложения (пусть это будет каталог C:\CarsWebSite). Следующее, что нам надо будет сделать, — создать на web-сервере новый виртуальный каталог, которому будет соответствовать этот физический каталог. Сделать это можно разными способами, но самый простой — в окне Internet Services Manager выбрать Default Web Site, щелкнуть на нем правой кнопкой мыши и в контекстном меню выбрать New (Новый) ▸ Virtual Directory (Виртуальный каталог) (рис. 14.2).

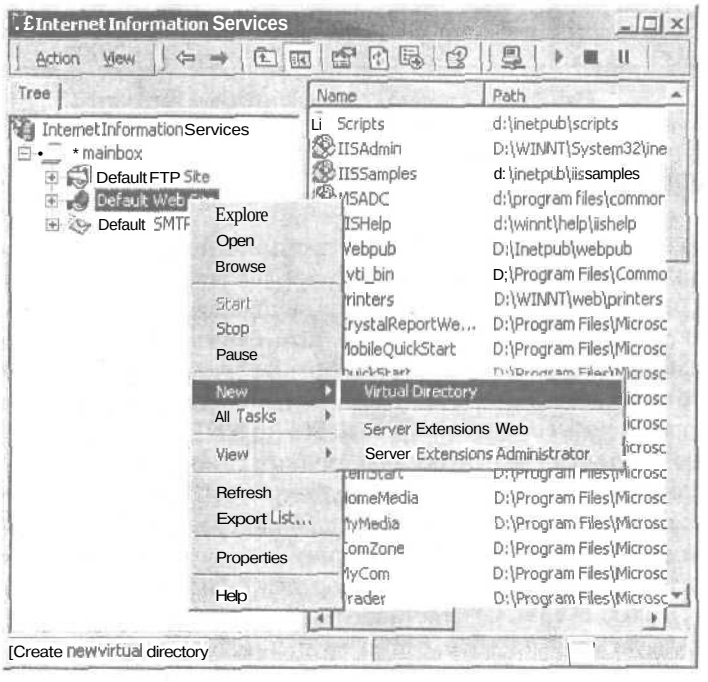


Рис. 14.2. Создаем виртуальный каталог

У нас запустится мастер создания виртуального каталога. Пропустим страницу с приветствием и присвоим создаваемому нами виртуальному каталогу имя Cars. Далее нас спросят о физическом пути в операционной системе для этого виртуального каталога. Выберем созданный нами каталог C:\CarsWebSite. Далее мастер задаст нам вопросы об основных параметрах нашего виртуального каталога (о возможности доступа к нему на чтение и запись, просмотра списка файлов из web-браузера, запуска скриптов и исполняемых файлов и т. п.). В нашем случае вполне подойдут значения, предлагаемые мастером по умолчанию (если нам потребуется что-либо изменить, это несложно будет сделать через свойства виртуального каталога). После того как все эти действия будут завершены, мы сможем увидеть созданный нами виртуальный каталог в списке каталогов web-сайта по умолчанию на сервере IIS (рис, 14.3).

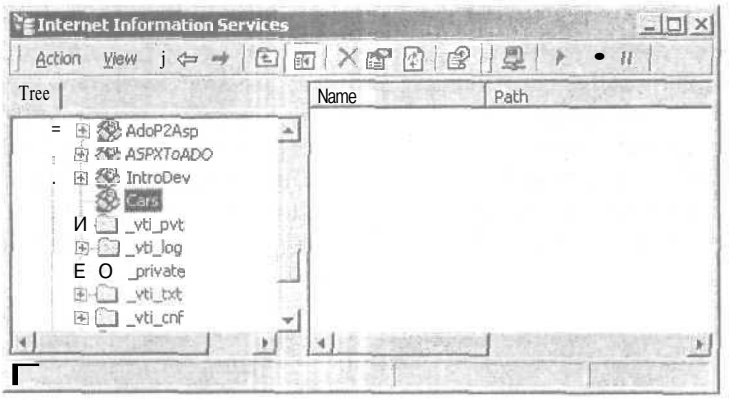


Рис. 14.3. Созданный нами виртуальный каталог

Структура документа HTML

Теперь, когда мы создали виртуальный каталог, можно приступать к созданию самого web-приложения. При создании web-приложений не обойтись без страниц на языке HTML. HTML (Hypertext Markup Language, язык гипертекстовой разметки) — это стандартный язык разметки, используемый для описания того, как текст, изображения, гиперссылки и стандартные элементы графического интерфейса будут отображаться в web-браузере. Большинство современных сред разработки web-приложений (в том числе Visual Studio.NET) позволяют создавать web-страницы, почти не обращаясь непосредственно к самому коду HTML, однако тем не менее разработчик web-приложений должен, безусловно, знать этот язык.

Документ HTML обычно начинается с набора тегов, в которых содержится общая информация о документе (заголовок, метаданные файла и г. п.), за которыми следует само тело документа (то есть набор текста, изображений, таблиц, гиперссылок и т. п.). Теги HTML не чувствительны к регистру: для браузера будет все равно, написали ли мы <HTML>, <html> или <Html>.

Давайте приступим к созданию документа HTML. Откроем интегрированную среду разработки Visual Studio.NET и в меню File (Файл) выберем Miscellaneous Files (Разные файлы) ▸ New File (Новый файл) и сохраним созданный файл в нашем физическом каталоге как default.htm. При этом Visual Studio.NET автоматически добавит в созданный нами файл HTML следующие теги:

```
<HTML>
<HEAD>
<TITLE></TITLE>
<META NAME="GENERATOR" Content="Microsoft Visual Studio">
<META HTTP-EQUIV="Content-Type" content="text/html">
</HEAD>
<BODY>

<!- Insert HTML here ·>

</BODY>
</HTML>
```

Открывающий тег HTML выглядит как <X>, а закрывающий — как </X>, хотя существует множество тегов, которые закрывать не надо. Теги <HTML> и </HTML> помечают начало и конец вашего документа HTML. Тег <HEAD> выделят метаданные для всего документа. В нашем случае внутрь блока <HEAD> помещены два тега <META>, которые описывают программу, использованную для создания этого документа (Microsoft Visual Studio), и содержимое файла. Пока у нашей страницы нет названия. Давайте его добавим:

```
<HTML>
<HEAD>
<TITLE>HTML is unavoidable</TITLE>
<META NAME="GENERATOR" Content="Microsoft Visual Studio">
<META HTTP-EQUIV="Content-Type" content="text/html">
</HEAD>
<BODY>

<!- Insert HTML here ->

</BODY>
</HTML>
```

То название, для которого мы использовали тег <TITLE>, выводится как заголовок окна браузера, в котором открыт наш документ (рис. 14.4).

Рис. 14.4. Применение тега <TITLE>

Само содержание документа HTML помещается между тегами <BODY> и </BODY>. Как правило, между этими тегами помещается множество дополнительных тегов, которые используются для представления и форматирования текстовой и графической информации. Все теги HTML мы, конечно, рассматривать не будем, но *самые* необходимые встретятся нам в этой главе не один раз.

Форматирование текста средствами HTML

Исходное назначение HTML заключалось в представлении текстовой информации. Как мы уже говорили, текст документа в HTML обычно помещается между тегами <BODY> и </BODY>. Например, предположим, что мы создаем страницу аутентификации пользователей. Текст HTML на ней может выглядеть следующим образом (обратите также внимание на синтаксис комментариев HTML):

```
<BODY>
      <!- Приглашение пользователю к аутентификации ->
      The Cars Login Page
</BODY>
```

В этом примере к нашему тексту не были применены какие-либо теги. Встречаясь с таким текстом, web-браузер выводит его в своем окне так, как он был записан, Если мы изменим текст документа следующим образом:

```
<BODY>
        <!- Приглашение пользователю к аутентификации ->
        The Cars Login Page
        Please enter your user name and password.
</BODY>
```

то браузер и не подумает добавить ожидаемый переход на новую строку (рис. 14.5).

Рис. 14.5. Для текстовой информации без тегов символы начала новых строк учитываться не будут

Чтобы начать новый абзац, необходимо выделить текст в этом абзаце при помощи тегов <P> и </P>, например, так:

```
<BODY>
        <!- Приглашение пользователю к аутентификации ->
        The Cars Login Page
        <P>Please enter your user name and password.</P>
</BOOY>
```

Теперь в окне браузера все выглядит по-другому (рис. 14.6).

Рис. 14.6. Тег <P> означает начало нового абзаца

Можно и не начинать новый абзац, а просто добавить теги начала новой строки
 и </BR>:

```
<BODY>
        <!- Приглашение пользователю к аутентификации ->
        The Cars Login Page
        <BR>Please enter your user name and password.</BR>
</BODY>
```

В этом случае браузер отобразит наш текст несколько иначе (рис. 14.7),

Рис. 14.7. Тег
 означает переход на новую строку

В HTML предусмотрены средства для выщеления участков текста полужирным шрифтом и курсивом. Для этого предусмотрены теги <B> </B> и <I> </I> соответственно:

```
<BODY>
        <!- Приглашение пользователю к аутентификации ->
        <B> The Cars Login Page </B>
        <BR>Please enter your <I>user name</I> and <I>password</I>.</BR>
</BODY>
```

Результат представлен на рис. 14.8.

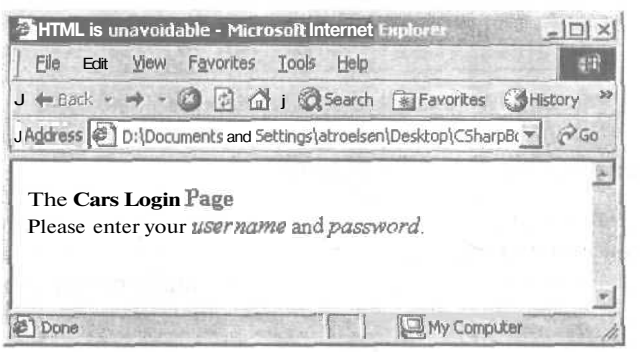


Рис. 14.8. Выделение текста полужирным и курсивным начертаниями

Заголовки HTML

Последний вид тегов для форматирования текста, который мы рассмотрим, — это теги заголовков HTML. Они выглядят как <H1>, <H2>, <H3>, <H4>, <H5> и <H6> и приме-

няются для изменения размера выделенного ими текста. Наибольший относительный размер текста обеспечивает тег <H!> (заголовок первого уровня). Вот пример:

```
<BODY>
        <!- Приглашение пользователю к аутентификации ->
        <H1> The Cars Login Page </H1>
        <BR><H3>Please enter your <I>user name</I> and <I>password</I>.<H3></BR>
</BODY>
```

Для того чтобы блок текста был выровнен посередине страницы, можно использовать тег <CENTER>:

```
<BODY>
        <!- Приглашение пользователю к аутентификации ->
        <CENTER>
        <H1> The Cars Login Page </H1>
        <BR><H3>Please enter your <I>user name</I> and <I>password</I>.<H3></BR>
        </CENTER>
</BODY>
```

То, как теперь выглядит наша страница, показано на рис. 14.9.

Рис. 14.9. Применение тегов заголовков и выравнивания

HTML-редактор Visual Studio.NET

Пока у нас получается очень простая, если не сказать примитивная, страница. Давайте мы ее немного оживим. Для этого нам потребуется добавить дополнительные теги HTML. Проще всего сделать это при помощи встроенных средств Visual Studio.NET.

Начнем с тех средств, которые применяются для управления отображением всего документа. Для этого выберем объект Document и откроем его свойства (рис. 14.10).

Например, если мы изменим значение свойства bgColor (background color — цвет фона), то в нашем документе HTML автоматически появится новый тег (рис. 14.11).

В Visual Studio.NET также предусмотрена панель форматирования HTML. С ее помощью можно управлять представлением блоков текста, выбирая для них цвет, шрифт, уровень заголовка, применение разметки списков и т. п. (рис. 14.12).

Рис. 14.10. Редактирование документа HTML при помощи графических средств Visual Studio.NET

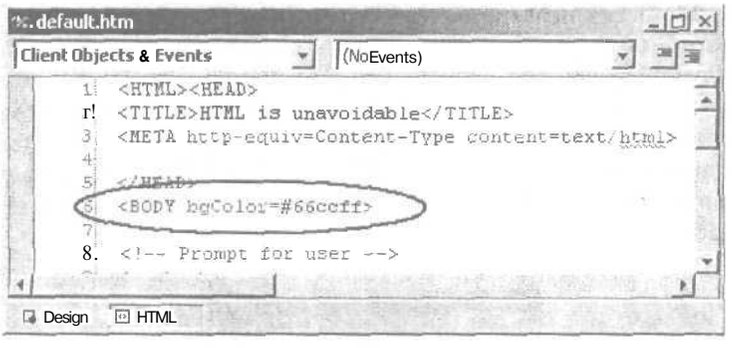


Рис. 14.11. Изменения, вносимые при помощи графических средств, сразу же появляются в коде HTML

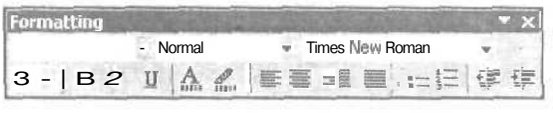


Рис. 14.12. Панель форматирования HTML

При помощи графических средств Visual Studio.NET мы можем оформить нашу страницу — весь необходимый для этого код HTML будет сгенерирован автоматически. Эти средства позволяют сэкономить много времени, однако web-разработчику часто приходится создавать весь код для страницы HTML вручную.

Разработка форм HTML

Будем считать, что с оформлением нашей страницы мы уже разобрались. Теперь настало время наделить ее новыми свойствами — возможностью принимать ввод

пользователя. Для этого нам придется прибегнуть к помощи элементов управления HTML. Как мы увидим дальше, в среде ASP.NET предусмотрен набор элементов управления WebForm, при применении которых все необходимые теги для элементов управления HTML будут генерироваться автоматически. Однако знакомство с тегами для создания элементов управления HTML также будет нелишним. Еще раз подчеркнем, что элементы управления WebForm в ASP.NET и элементы управления HTML — это разные вещи, и в процессе выполнения web-приложения элементы управления WebForm преобразуются в элементы управления HTML,

Форма HTML — это именованная группа элементов пользовательского интерфейса HTML, используемых для ввода пользователем данных. Затем эти данные передаются на web-сервер по протоколу HTTP (подробнее об этом — чуть позже). Теги для элементов пользовательского интерфейса на форме HTML помещаются между тегами <form> и </form>:

```
<form name = MainForm id = MainForm>
      <!- Add UI elements here ->
</form>
```

В этом коде мы создали форму и присвоили ей, во-первых, дружественное имя, а во-вторых, идентификатор. С технической точки зрения использовать имя в принципе не обязательно, однако во многих ситуациях это очень удобно.

Как правило, в открывающий тег <form> помещается атрибут для действия, выполняемого этой формой. В нем содержится информация об адресе URL, на который будут передаваться данные, введенные пользователем, а также сведения о методе передачи данных (POST или GET). Мы вскоре рассмотрим эти моменты достаточно подробно, а пока давайте рассмотрим те элементы, которые могут быть помещены внутрь формы HTML. В Visual Studio.NET предусмотрена специальная панель HTML Toolbox, в которой мы можем выбрать эти элементы (рис. 14.13).

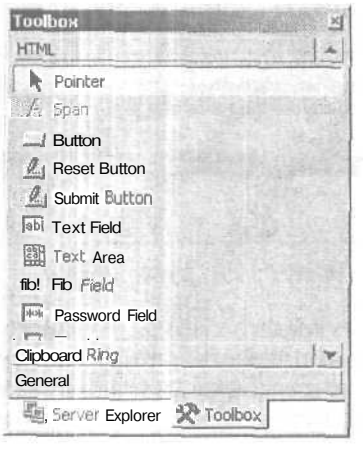


Рис. 14.13. Элементы управления HTML

Краткий перечень наиболее часто используемых элементов представлен в табл. 14.1.

Таблица 14.1. Элементы управления HTML

Элемент управления	Описание
Button	Эта разновидность кнопки обычно используется для того, чтобы выполнить отрезок кода клиентского скрипта. Для отправки данных на web-сервер используется специальная кнопка Submit Button, а для возврата формы в исходное состояние — Reset Button
Checkbox Radio Button Listbox **Dropdown**	То же самое, что и аналогичные элементы управления Windows Forms
Image	Позволяет указать изображение, которое будет выведено на форме
Reset Button	Специальная кнопка на форме, при нажатии на которую все значения в форме принимают свой исходный вид
Submit Button	Еще одна специальная кнопка, при нажатии на которую производится отправка данных формы на web-сервер
Text Field Text Area	Эти элементы управления предназначены для ввода пользователем одной строки текста или нескольких строк
Password Field	Специальное текстовое поле, предназначенное для ввода пользователем пароля. Все символы в этом поле отображаются звездочками

В библиотеке базовых классов .NET предусмотрен набор типов .NET, которые соответствуют элементам управления HTML. Они определены в пространстве имен System.Web.UI.HtmlControls.

Создаем пользовательский интерфейс

Первое, что нужно сделать, чтобы наша страница могла воспринимать ввод пользователя, — создать на ней форму **HTML.** Для этого поместим на страницу следующий код:

```
<HTML>
<HEAD>
<TITLE>HTML is unavoidable</TITLE>
<META NAME="GENERATOR" Content = "Microsoft Visual Studio">
<META HTTP-EQUIV="Content-Type" content="text/html">
</HEAD>
<BODY BGCOLOR="66ccff">
<!- Приглашение пользователю к аутентификации ->
<center>
<h1>The Cars Login Page</h1>
<br><h3>Please enter your <i>user name</i> and <i>password</i>.</h3>

<!- Создаем форму для ввода пользователем информации ->
<form name=MainForm>
</form>

</center>
</BODY>
</HTML>
```

Форма создана, теперь можно приступать к добавлению в нее элементов управления. Это можно сделать при помощи графических средств Visual Stu-

dio.NET, а можно создать все необходимые теги вручную. Каждый элемент управления описывается атрибутом имени (имя используется при выполнении программы, чтобы определить, в какой элемент управления были введены данные) и атрибутом типа (этот атрибут и определяет разновидность элемента управления). Для разных элементов управления существуют разные наборы дополнительных атрибутов, которые могут быть использованы для определения различных их параметров. Конечно же, эти дополнительные параметры можно также настроить при помощи окна свойств для этого элемента управления в Visual Studio.

Наша форма будет содержать два текстовых поля (одно — для ввода имени пользователя, другое, специальное парольное — для ввода пароля) и две кнопки — для передачи информации на сервер и для восстановления формы в исходном состоянии, если пользователь решил отменить свой ввод. Вот код HTML для нашей формы (" " определяет вставку символа пустого пространства);

```
<form name=MainForm>
      .<p>User Name:  
       <input id = txtUserName type * text></p>
       <p>Password:   
       <input name = txtPassword type = password></p>
       <input name = btnSubmit type = submit value = Submit>  
       <input name = btnReset type = reset value = Reset>
</form>
```

Для каждого элемента управления мы определили уникальное имя (txtUserName, txtPassword, btnSubmit и btnReset). Кроме того, для каждой кнопки мы определили очень важный атрибут value (значение). value = Reset означает, что все элементы управления на форме вернутся в исходное состояние, а value = Submit — что данные, введенные пользователем, отправятся получателю.

Атрибут value можно применять не только для кнопок. Например, мы можем определить атрибут value для текстового поля txtUserName (рис. 14.14).

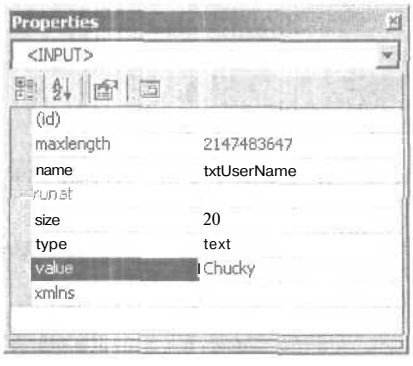


Рис. 14.14. Установка атрибута value для элемента управления

Если мы определим для txtUserName атрибут value = Chucky, то это значит, что слово Chucky станет значением по умолчанию для этого текстового поля и оно будет помещаться в поле всякий раз при загрузке формы (рис. 14.15).

Рис. 14.15. Применение значения по умолчанию для текстового поля

Добавление изображений

Последняя тема, посвященная возможностям HTML, которую мы рассмотрим, — это добавление на web-страницу изображений. Изображения добавляются при помощи тегов <img>:

```
<img alt="You gotta log in to see this!" src="car.gif" border=4>
```

Атрибут <alt> (от alternative) используется для определения текстового эквивалента изображения. Этот текст «всплывет», если поместить указатель мыши над изображением, или будет выведен вместо самого изображения, если браузер не поддерживает вывод графики. Необязательный атрибут border определяет толщину рамки вокруг изображения. В атрибуте src (от source — источник изображения) можно использовать как полный путь к файлу изображения, так и относительный путь, при котором подразумевается, что файл изображения будет находиться в одном каталоге с файлом *.htm. Наша страница после обновления будет выглядеть так, как показано на рис, 14.16

Клиентские скрипты

Одной из больших проблем для множества web-приложений является необходимость вновь обращаться с повторными запросами на web-сервер для внесения изменений в то, что показывается пользователю в окне браузера. Конечно, во многих случаях такие обращения неизбежны, но если есть возможность сократить их количество, то этой возможностью надо пользоваться. Один из способов сократить количество запросов на web-сервер заключается в применении клиентских (браузерных) скриптов, например, для проверки введенных пользователем данных перед передачей этих данных на сервер.

Рис, 14.16. Web-страница после добавления изображения

Например, в нашей ситуации пользователю необходимо ввести имя пользователя и пароль для аутентификации. Если какое-либо из полей останется незаполненным, аутентификация все равно не произойдет. Поэтому вполне можно сделать так, чтобы пользователь не мог отправить данные на сервер, не заполнив оба поля. Конечно, в этой ситуации только кодом HTML нам не обойтись: HTML — это язык разметки, а не программирования. Для того чтобы реализовать проверку введенных пользователем данных, нам придется использовать какой-либо из языков для работы со скриптами.

Существует множество языков для работы со скриптами, но для скриптов, выполняющихся в браузерах, подойдут далеко не все. Internet Explorer поддерживает два языка: VBScript (диалект Visual Basic для работы со скриптами) и JavaScript, а Netscape Navigator — только JavaScript. Если мы можем гарантировать, что клиенты нашего приложения будут использовать только Internet Explorer, можно использовать любой из языков для работы со скриптами. Но на обычных web-сайтах, открытых для доступа самых разных клиентов, как правило, используется только один язык для браузерных скриптов — JavaScript.

JavaScript — очень популярный язык скриптов, который де-факто является стандартом для создания браузерных скриптов. Сразу заметим, что JavaScript никоим образом не является частью языка Java. JavaScript — это совершенно отдельный язык программирования, который предназначен для решения специальных задач и в котором предусмотрено гораздо меньше возможностей, чем в Java. JScript — это название реализации JavaScript от Microsoft.

Пример клиентского скрипта

Как правило, клиентские скрипты выполняются в ответ на события графических элементов HTML. Как же происходит перехват таких событий? Проще всего показать это на примере.

Предположим, что мы работаем с очень простой web-страницей, представленной на рис. 14.17. Единственная кнопка на этой странице будет называться testBtn (проще всего присвоить это имя из свойств кнопки). Чтобы настроить перехват события, возникающего при нажатии этой кнопки, перейдем в режим просмотра HTML и выберем нашу кнопку в левом ниспадающем списке. Затем в правом списке выберем для этой кнопки событие onclick (рис. 14.18).

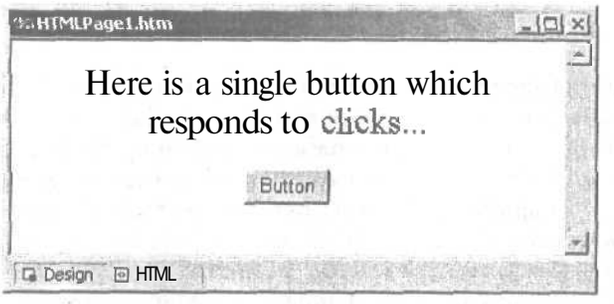


Рис. 14.17. Новая страница HTML

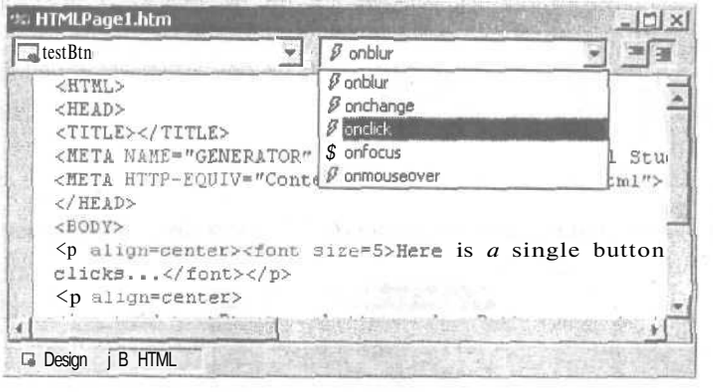


Рис. 14.18. Перехват событий элементов управления HTML

Выполнив эти действия, мы сможем обнаружить в коде HTML следующие изменения:

```
<HTML>
<HEAD>
<TITLE></TITLE>
<META NAME="GENERATOR" Content="Microsoft Visual Studio">
<META HTTP-EQUIV="Content-Type" content="text/html">

<Script ID=clientEventHandlerJS language=javascript>
```

```
<!-
function testBtn_onclick() {
}
//->
</script>

</HEAD>
<BODY>
<p align = center>
<font size - 5>Here is a single button which responds to clicks...</font></p>
<p align = center>
<1nput id =testBtn type-button value=Button name=testBtn language=javascript
                                              onclick="return testBtn_onclick()">
</p>

/BODY>
/HTML>
```

Как мы видим, на нашей странице в разделе <head></head> появился блок <script>, для которого в качестве используемого языка указан JavaScript. Обратите внимание, что сам код скрипта помещен в блок комментария HTML. Причина понятна — если страница будет открыта в браузере, который не поддерживает JavaScript, то этот код будет воспринят как комментарий и проигнорирован.

Обратите также внимание, что в теге для нашей кнопки появился новый атрибут onclick, который ссылается на метод JavaScript. В результате при нажатии на эту кнопку будет вызван этот метод. Сильно усложнять содержание метода мы не будем, и для наших целей его код будет таким:

```
<script id = clientEventHandlersJS language = javascript>
<!-
function testBtn_onclick()
{
        // Аналог MessageBox в JavaScript
        alert"Hey, stop clicking me...");
}
//->
</script>
```

При нажатии на кнопку мы получим окно сообщения, представленное на рис. 14.19.

Рис. 14.19. Окно оповещения (alert) Internet Exlorer

Реализация проверки введенных пользователем данных

Давайте теперь займемся более сложной ситуацией и реализуем проверку ввода пользователя на нашей странице default.htm. Задача проста: при нажатии на кнопку Submit должен вызываться метод JavaScript, который будет проверять, не оставлено ли какое-либо из текстовых полей пустым. Если так оно и есть, пользователю

будет выдаваться сообщение Internet Explorer с информацией о допущенной им ошибке. Прежде всего нам потребуется определить для кнопки Submit событие onclick. При возникновении этого события должен вызываться метод JavaScript ValidateData(). Этот метод будет проводить проверку на отсутствие данных в текстовых полях:

```
<script language = javascript>
<!- Необходимо использовать полные имена текстовых полей в формате имя_формы.имя_поля!
function ValidateDataO
{
        // Если что-то забыто, выводим окно сообщения
        if((MainForm.txtUserName.value = = "") || (MainForm.txtPassword.value = = ""))
        {
                alert("You must supply a user name and password!");
                return false:
        }
        return true:
}
->
...
<input id = btnSubmit onclick - "return ValidateDataO" type=submit value = Submit name
                                                                = btnSubmit>
```

С проверкой данных в нашей форме все. Однако в этом примере были продемонстрированы лишь самые примитивные возможности JavaScript. Чтобы дать хотя бы небольшое представления о других возможностях этого языка и браузерных скриптов в целом, давайте создадим еще одну функцию, которая будет вызываться при загрузке страницы и выводить информацию о дате и времени входа пользователя. Для этой функции (она будет называться GetTheDate()) нам потребуется еще один тег <scr1pt>. Обратите внимание на применение метода write() объекта Document, представляющего текущий документ, загруженный в Internet Explorer.

```
<HTML>
<HEAD>
<TITLE>HTML is unavoidable</TITLE>
<scr1pt language = javascript>
<!- Методы JavaScript для этой формы
function ValidateDataO
{
        if((MainForm.txtUserName.value = = "") |[ (MainForm.txtPassword.value = - ""))
        {
                alert("You must supply a user name and password!");
                return false:
        }
        return true;
}
function GetTheDate() { return Date(); }
->
</script>

</HEAD>
<BODY bgColor=#66ccff>

<!- Приглашение пользователю для ввода ·->
<center>
<h1>The Cars Login Page</hl>
```

```
<h2>Today is: </h2>
<script language=javascript>
        // Метод document.write - часть объектной модели IE
        document.write(GetTheDate());
</script>
<br>
</h3>Please enter your <i>user name</i> and <i>password</i>.</h3>

<!- Создает форму для приема ввода данных пользователей ->
<form name=MainForm>
        <p>User Name:   <input type=text name=txtUserName></p>
        <p>Password:    <input type=passord name=txtPassword> </p>
        <input id=btnSubmit onclck=ValidateData() type-submit value=Submit
                                                name=btnSubmit>  
        <input type=reset value-Reset name=btnReset>
</form>
</center>
</BODY>
</HTML>
```

Передаем данные формы (методы GET и POST)

Будем считать, что с вводом данных пользователем и их проверкой мы уже разобрались. Однако введенные и проверенные данные обычно необходимо отправить получателю — например, другому файлу HTML на web-сервере, почтовому серверу, странице ASP (Active Server Page) и т. п. Получатель данных формы указывается при помощи атрибута action. Давайте добавим этот атрибут в тег нашей формы:

```
<form name=MainForm
action="http://localhost/Cars/ClassicASPPage.asp" method - "GET">
...
</form>
```

Такое значение атрибута action означает, что при нажатии кнопки Submit данные формы будут переданы странице ASP с именем ClassicASPPage.asp, расположенной в нашем виртуальном каталоге Cars на локальном компьютере. Мы определили в качестве метода для передачи данных метод GET. Это значит, что данные формы будут добавлены к запросу в виде пар имя — значение. Вместо метода GET можно использовать метод POST, в этом случае значение соответствующего атрибута должно выглядеть следующим образом:

```
<form name=MainForm
action="http://localhost/Cars/ClassicASPPage.asp" method - "POST">
...
</form>
```

При использовании метода POST данные формы не будут добавляться к строке запроса. Вместо этого для них будет выделена отдельная строка в поле заголовка HTTP. При этом передаваемые данные не будут сразу же видны всем, кто отслеживает наш трафик, и поэтому метод POST можно считать немного более безопасным. Однако еще важнее, что при помощи метода POST можно передавать данные гораздо большего размера, чем при помощи GET. Однако в нашем примере мы будем использовать более традиционный и простой метод GET.

Синтаксис строки запроса HTTP

Файл ASP, которому мы передаем данные из нашей формы, должен суметь извлечь эти данные из строки запроса. Сама строка запроса с данными формы выглядит как обычный адрес в адресной строке браузера с добавлением нескольких пар имя — значение;

```
http://localhost/Cars/
ClassicAspPage.asp?txtUserName=Chucky&txtPassword=somepassword&btnSubmit=Submit
```

Обратите внимание, что вся строка запроса разбивается на две части символом знака вопроса (?). Слева от знака вопроса находится адрес получателя данных, а справа — сами пары имя — значение (например, txtUserName=Chucky).

Как можно убедиться, каждая пара имя — значение отделена от другой пары символом амперсанда (&). Строка в нашем примере не представляет никаких сложностей для анализа, поскольку передаваемые значения очень просты. Однако, к примеру, если бы нам потребовалось поместить внутрь какого-либо значения пробел (Chucky заменить на Chucky Chuckles), то строка запроса выглядела бы уже следующим образом:

```
http://localhost/Cars/
ClassicAspPage.asp?txtUserName=Chucky+Chuckles&txtPassword=somepassword&btnSubmit=Submit
```

Таким образом, вместо пробелов в передаваемых значениях подставляется символ +. Если нам придет в голову поставить между Chucky и Chuckles пять пробелов, то строка запроса будет выглядеть так:

```
http://localhost/Cars/
ClassicAspPage.asp?txtUserName=Chucky+++++Chuckles&txtPassword=somepassword&btnSubmit=Submit
```

А что будет, если в передаваемом значении попадутся какие-нибудь служебные символы? Они будут переданы в виде символа процента, за которым следует шестнадцатеричное значение соответствующего символа ASCII. Например, если пользователь придумал себе пароль, который выглядит как Hello^77, то этот пароль будет передан так:

```
http://localhost/Cars/
ClassicAspPage.asp?txtUserName=Chucky+++++Chuckles&txtPassword=Hello%5E77&btnSubmit=Submit
```

Создание классической страницы ASP

Наша форма уже в состоянии передавать данные, однако принимать их пока некому. Давайте ликвидируем этот недостаток нашего web-приложения и создадим страницу — приемник наших данных. Вначале мы создадим ее не на ASP.NET, а с помощью классических ASP.

Первое, с чего нам нужно начать — добавить новую страницу Active Server Page при помощи Visual Studio.NET (рис. 14.20). Присвоим этой странице то имя, которое мы указали в атрибуте action для формы (ClassicASPPage.asp) и сохраним ее в физическом каталоге, которому соответствует виртуальный каталог Cars.

Страницу ASP можно воспринимать как набор из кода HTML и скриптов, предназначенных для выполнения на сервере. Можно сказать, что основной смысл ASP заключается в генерации кода HTML «на лету» при помощи серверных скриптов.

Например, можно создать страницу ASP, которая будет считывать данные из источника данных (при помощи ADO) и представлять возвращаемые строки в виде кода HTML.

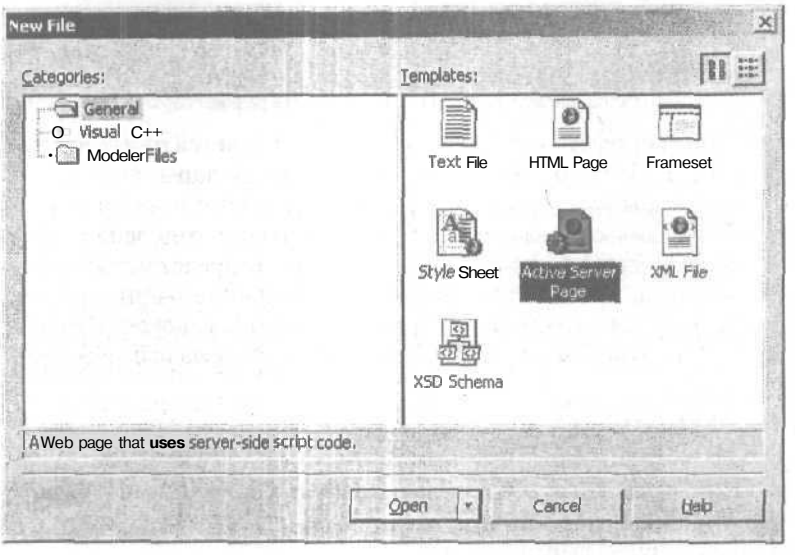


Рис. 14.20. Добавление классического файла ASP

В нашем примере страница ASP будет использовать встроенный объект Request для считывания значений из входящей строки запроса и выводить полученные от клиента данные в виде кода HTML. Можно сказать, что клиенту будет возвращаться эхо его запроса. Вот код соответствующего скрипта (обратите внимание на блок <%...%>, в который помещен скрипт):

```
<%@ Language=VBScript %> <!- VBScript вполне подходит для серверных скриптов ->
<HTML>
<HEAD>
<META NAME="GENERATOR" Content-Microsoft Visual Studio 7.0">
</HEAD>
<BODY>
<!• Возвращаем обратно то. что получили ->
<center>
        <h1>You said: </hl>
        <b>User Name: </b><%= Request.QueryString("txtUserName") %><br>
        <b>Password: </b><%= Request.QueryString("txtPassword") %><br>
</center>
/<BODY>
</HTML>
```

Первое, что необходимо отметить, — на странице ASP используются те же теги <HTML>, <HEAD> и <BOQY>, что и на обычной web-странице. Мы используем объект Request, который, как и положено объектам, поддерживает некоторое количество свойств, методов и событий. Для того чтобы извлечь данные в виде запроса от клиента, используется метод Request .QueryString().

Как же генерируется код HTML, который будет возвращен клиенту? Можно сказать, что запись <%...%> означает: «Вставь это в HTTP-ответ». Кроме того, мы можем получить полный контроль над тем, что возвращается пользователю, при помощи объекта Response. Например:

```
<!- Возвращаем обратно то. что получили ->
<center>
        <hl>You said: </hl>
        <b>User Name: </b><%= Request.QueryString("txtUserName") %><br>
        <b>Password: </b>
        <%
                dim pwd
                pwd = Request.QueryString("txtPassword")
                Response.Write(pwd)
        %>
</center>
```

В типах Request и Response, конечно же, предусмотрено множество других очень полезных членов, кроме того, в распоряжении web-разработчика классических ASP также находится набор дополнительных объектов, таких как Session, Server, Application и ObjectContext. Мы не ставим своей целью в этой главе разобрать классические ASP во всех подробностях, поэтому эти объекты мы рассматривать не будем (информацию о них можно получить в электронной документации по Visual Studio). Однако отметим, что возможности этих типов реализованы в ASP.NET при помощи типа Page.

Для того чтобы запустить нашу страницу ASP, просто откроем файл default.htm, введем в текстовые поля значения для имени пользователя и пароля и нажмем кнопку Submit Сработает серверный скрипт ASP, и в окне браузера откроется сгенерированная на основе ваших данных страница (рис. 14.21).

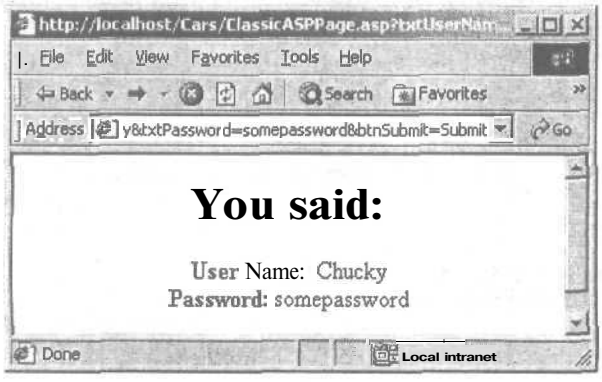


Рис. 14.21. Динамически созданная страница HTML

Наш пример, конечно, трудно отнести к разряду очень изощренных, но он хорошо иллюстрирует основной принцип работы ASP (и ASP.NET): данные передаются через форму HTML, обрабатываются серверным скриптом, и результат возвращается пользователю в виде сгенерированного кода HTML.

Принимаем данные, переданные методом POST

В нашем примере для передачи данных формы использовался метод GET, при котором пары имя — значение для элементов управления формы добавлялись к концу строки запроса. Затем значения принимались при помощи метода Request.QueryString(). Сразу отметим, что этот метод может использоваться только для приема значений, передаваемых методом GET. Если мы изменим значение соответствующего тега формы на POST и снова запустим наше приложение, ничего хорошего не произойдет: нам вернутся пустые значения (рис. 14.22).

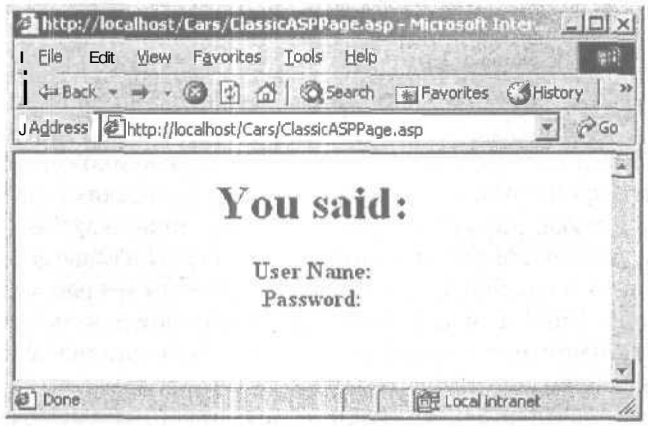


Рис. 14.22. Метод QueryString() может принимать только информацию, переданную при помощи метода GET

Конечно же, в типе Request предусмотрены члены, которые позволяют принимать данные, отправленные и методом POST. Для этой цели используется коллекция Form. Выглядит это так:

```
<BODY>
<!- Возвращаем обратно то, что получили •>
<center>
        <h1>You said: </h1>
        <b>User Name: </b><%= Request.Form("txtUserName") %><br>
        <b>Password: </b>
        <%
                dim pwd
                pwd = Request.Form("txtPassword")
                Response.Write (pwd)
        %>
</center>
</BODY>
```

Давайте изменим код страницы ASP в соответствии с вышеприведенным примером и запустим наше web-приложение заново. Все работает! Результат может быть таким, как показано на рис. 14.23. Обратите также внимание, что теперь в адресной строке браузера переданные нами значения не отображаются.

Общая схема работы приложения ASP при использовании различных методов передачи данных представлена на рис. 14.24.

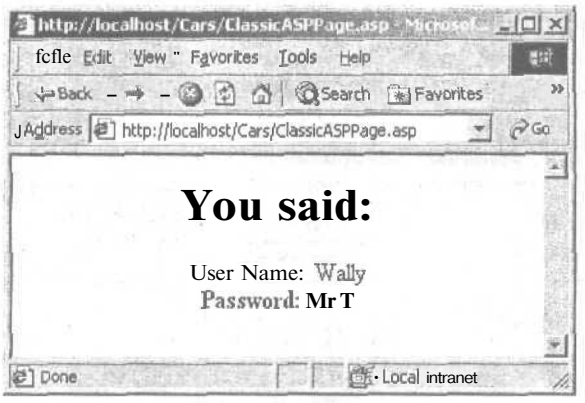


Рис. 14.23. Данные, переданные методом POST, принимаются при помощи Request.Form

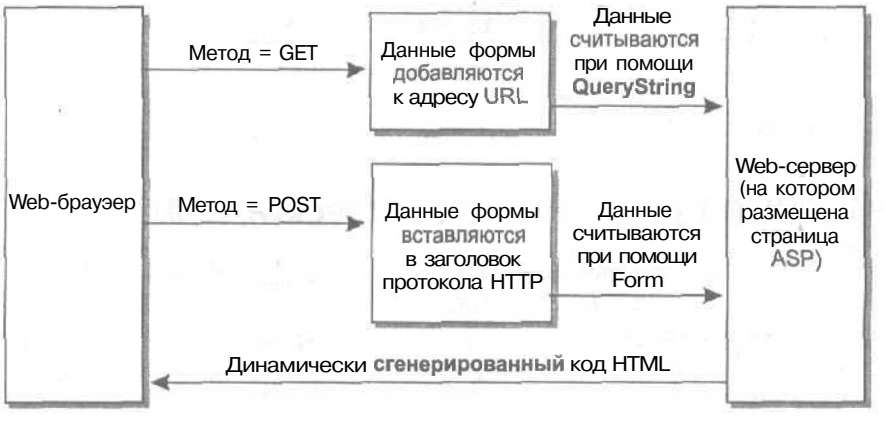


Рис. 14.24. Передача данных на страницу ASP при помощи методов GET и POST

Первое приложение ASP.NET

Перед тем как завершить эту вводную часть, давайте произведем следующую операцию: откроем файл default.htm и внесем в него следующие изменения (не забудем добавить к расширению файла ASP букву x):

```
<form name=MainForm
action="http://localhost/Cars/ClassicASPPage.aspx"
method=post ID=Form1>
```

Затем изменим расширение нашего файла ASP на *.aspx и запустим наше приложение снова. Как можно убедиться, все будет работать так же, как и раньше (рис. 14.25).

Нас можно поздравить! Мы только что создали первое приложение ASP.NET. Это было не очень сложно — достаточно было изменить расширение файла классических ASP с *.asp на *.aspx. Вывод напрашивается сам собой — то, что мы использовали в нашем файле классических ASP, используется и в ASP.NET. Одна-

ко, конечно же, в ASP.NET достаточно много отличий от технологии классических ASP. Многие из этих различий (как и общие возможности ASP и ASP.NET) мы рассмотрим в оставшейся части этой главы.

Рис. 14.25. Приложение ASP.NET

Код приложения Cars можно найти в подкаталоге Chapter 14.

Некоторые проблемы классических ASP

Классические ASP — это очень популярная архитектура создания web-приложения, однако она не лишена недостатков. Главный недостаток классических ASP заключается в том, что в них используются языки скриптов. Несмотря на всяческие хитрые приемы (например, кэширование откомпилированных скриптов для более быстрого повторного выполнения), языки скриптов — это большой проигрыш как в производительности (поскольку они являются интерпретируемыми), так и в возможностях (поскольку в них не поддерживаются многие технологии объектно-ориентированного программирования).

Еще одно неудобство классических ASP связано с тем, что в них код HTML смешан с кодом скриптов. В принципе, классические ASP позволяют размещать код HTML отдельно от кода скриптов, но суть дела от этого не меняется: логика представления (код HTML) не отделена от бизнес-логики (то есть от собственно исполняемого кода).

Еще один момент, знакомый любому web-разработчику, использующему ASP, заключается в том, что из проекта в проект приходится переносить одни и те же повторяющиеся блоки кода скриптов. В большинстве web-приложений требуется выполнять одни и те же действия: проверять данные, вводимые пользователем, обеспечивать форматирование HTML и т. п. Гораздо удобнее было бы использовать уже готовые решения, а не копировать код скриптов из одного проекта в другой.

Некоторые преимущества ASP.NET

В ASP.NET устранены многие недостатки классических ASP. Например, в файлах ASP.NET (*.aspx) языки скриптов не используются. Вместо этого мы можем при-

В ASP.NET для каждого открытого сеанса хранится своя уникальная информация, представленная при помощи типа HttpSessionState. Можно сказать, что каждому пользователю выделена область оперативной памяти, в которой хранятся промежуточные результаты его взаимодействия с web-приложением. Например, один пользователь, подключившийся к нашему web-приложению Cars, может интересоваться информацией о новом BMW, а другой — о Colt производства 1970 года. Отношения между web-приложением и сеансами подключения к нему представлены на рис. 14.26.

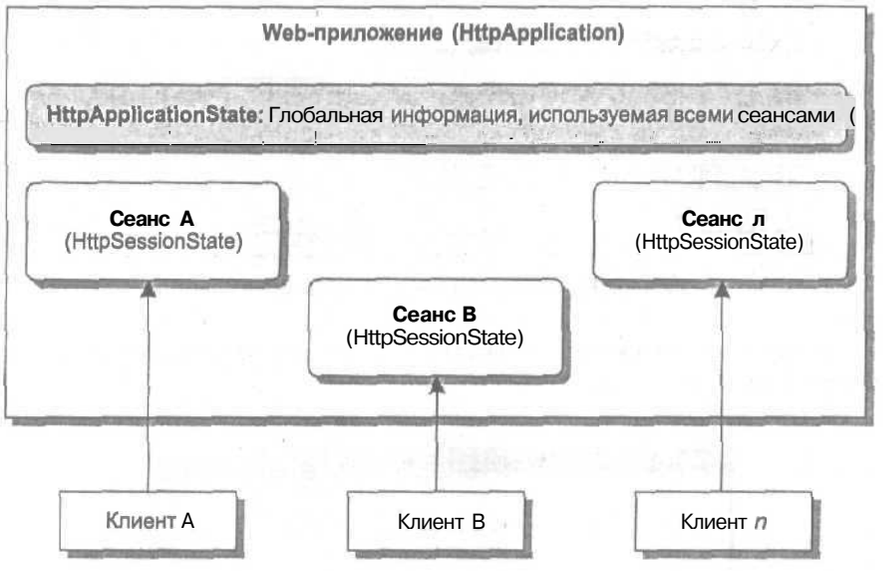


Рис. 14.26. Приложения и сеансы подключения

В классических ASP понятия приложения и сеанса представлены двумя отдельными типами (Application и Session). В ASP.NET они представлены вложенными типами HttpApplicationState и HttpSessionState, доступ к которым производится через свойства Application и Session типов, производных от Page. Мы вскоре поработаем с этими типами.

Создание простого web-приложения на *C#*

Давайте создадим маленький пробный проект, на примере которого познакомимся с основными принципами архитектуры ASP.NET.

Первое, что нужно сделать, — создать новый проект C#, выбрав для него шаблон Web Application (рис. 14,27). Мы назовем этот проект FirstWebApplication.

Перед тем как нажать кнопку OK, обратим внимание на то, что в поле Location представлен путь не к каталогу на жестком диске, как обычно, а адрес URL компьютера, на котором расположено web-приложение. Файлы решения Visual Studio.NET (*.sln и *.suo) будут помещены в каталог My Documents\Visual Studio Projects.

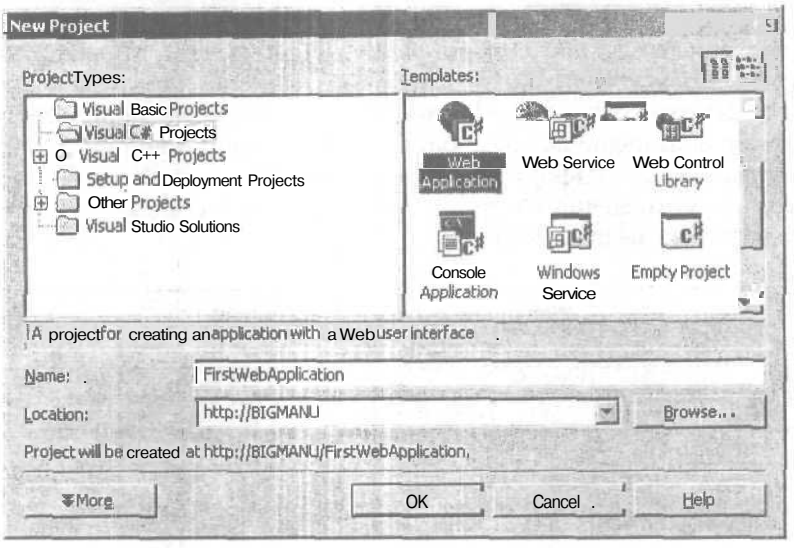


Рис. 14.27. Создание приложения ASP.NET на C#

После того как создание проекта будет завершено, перед нами откроется шаблон времени разработки (рис. 14.28).

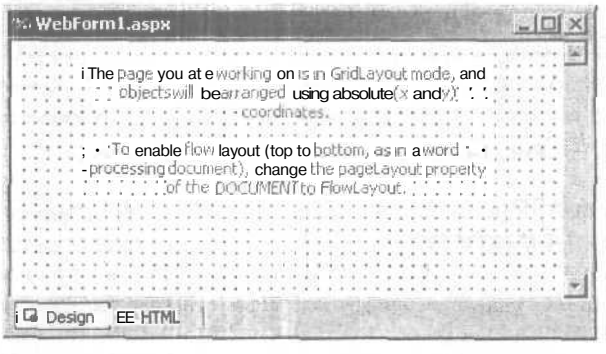


Рис. 14.28. Шаблон времени разработки web-приложения

Этот шаблон действует, как обычный шаблон Windows Forms, представляя графический интерфейс создаваемого нами файла *.aspx. Главное отличие заключается в том, что мы используем элементы управления не Windows Forms, а Web Form Controls, основанные на коде HTML. Обратите также внимание, что по умолчанию этой странице присвоено имя WebForm1. Учитывая, что к этой странице будут обращаться «из внешнего мира», лучше переименовать ее в default.aspx.

В окне Solution Explorer мы можем заметить, что по сравнению с обычным приложением добавилось множество ссылок на внешние сборки и дополнительные файлы (рис. 14.29).

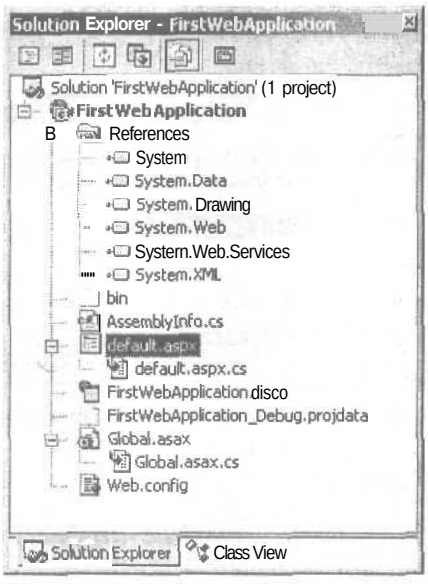


Рис. 14.29. Исходный вид web-приложения в окне Solution Explorer

Если же мы откроем Internet Services Manager, то сможем убедиться, что на сервере IIS появился новый виртуальный каталог FirstWebApplication (рис. 14.30).

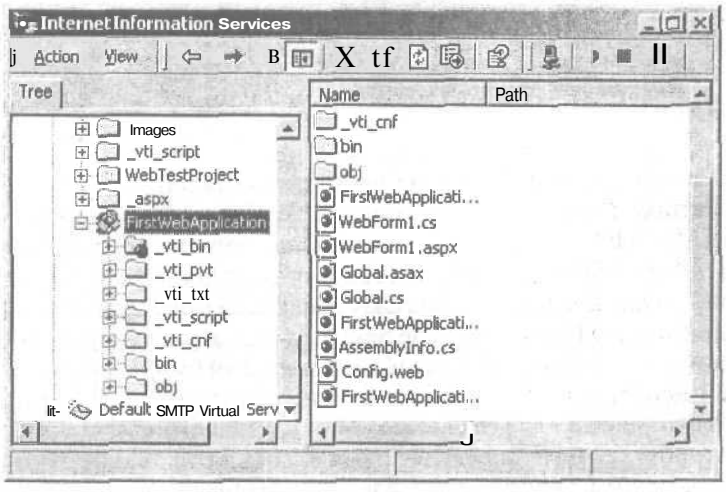


Рис. 14.30. При создании проекта Web Application на сервере IIS автоматически создается новый виртуальный каталог

Как можно убедиться, каждый файл, который мы добавим в наш проект, будет помещен в этот виртуальный каталог. Физически этому виртуальному каталогу будет соответствовать каталог FirstWebApplication в подкаталоге <имя_диска>:\Inetpub\wwwroot (рис. 14.31).

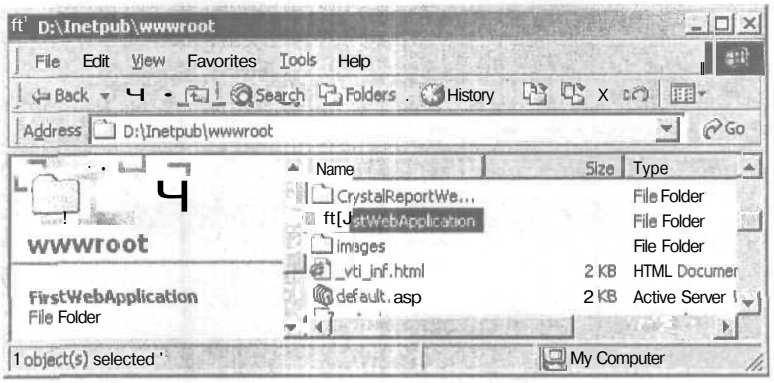


Рис. 14.31. Физический каталог, в котором расположены файлы вашего проекта

Исходный файл *.aspx

Если мы откроем автоматически сгенерированный файл *.aspx, то найдем в нем минимальный набор тегов с единственной формой:

```
<%@ Page language="c#" Codebehind="defaul.aspx.cs"
AutoEventWireup="false" Inherits="FirstWebApplication.WebForm1" %>
<HTML>
        <HEAD>
                <meta name=vs_targetSchema content="Internet explorer 5.0">
                <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
                <meta name="CODE_LANGUAGE" Content="C#">
        </HEAD>
        <body MS_POSITIONING="GridLayout">
                <form raethod="post" runat="server">
                </form>
        </body>
</HTML>
```

В этом коде привлекают внимание несколько деталей. Во-первых, обратите внимание на атрибут runat в открывающем теге <form>. Этот атрибут — один из важнейших в ASP.NET. Он означает, что данный элемент должен быть обработан средой выполнения ASP.NET, которая вернет результат браузеру клиента.

Кроме того, в коде предусмотрено сразу несколько моментов, относящихся ко всей странице в целом. В самом начале используется атрибут language. Его значение определяет, что для создания кода HTML, который будет возвращен браузеру клиента, будет использован C#. Атрибут Codebehind определяет имя файла C#, который будет использован для всех вычислений «за сценой» (behind означает «за», «позади»). Атрибут Inherits определяет имя класса, представляющего класс, определенный в Codebehind. Если что-то осталось непонятным, не волнуйтесь, мы вскоре поработаем с этими атрибутами.

Файл web.config

Файл web.config — это текстовый файл в формате XML, который используется для определения множества параметров нашего web-приложения. Обычно этот файл расположен в корне виртуального каталога и используется для каждого подка-

талога. По умолчанию в него помещается информация о компиляции, ошибках, безопасности, отладке и глобализации (рис. 14.32). Кроме того, в него могут быть помещены и другие необходимые нам данные.

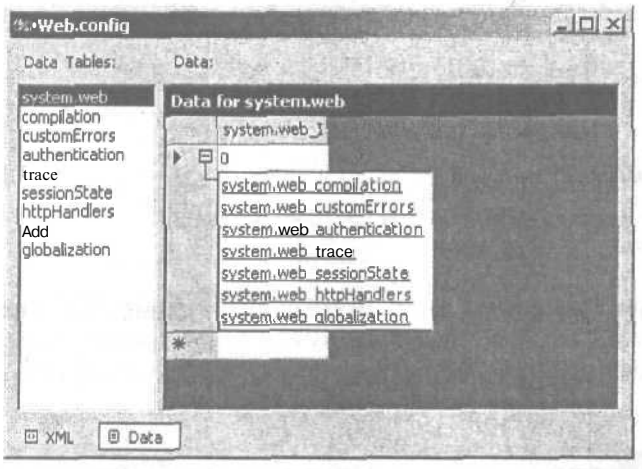


Рис. 14.32. Файл web.config позволяет настроить основные параметры нашего web-приложения

Исходный файл Global.asax

Как и в классических ASP, в ASP.NET используется глобальный файл (global.asax), который позволяет взаимодействовать с событиями как уровня всего приложения, так и уровня сеанса подключения. Кроме того, этот файл делает возможным совместное использование различных общих данных. Если мы щелкнем на этом файле в окне Solution Explorer правой кнопкой мыши и в контекстном меню выберем View Code, то сможем просмотреть эту информацию, которая представлена при помощи класса Global. Этот класс является производным от базового класса HttpApplicatiэл:

```
public class Global : System.Web.HttpApplication
{
        protected void Application_Start(Object sender. EventArgs e) {}
        protected  void Session_Start(Object sender. EventArgs e) {}
        protected void Application_BeginRequest(Object sender, EventArgs e) {}
        protected void Application_EndRequest(Object sender, EventArgs e) {}
        protected void Session_End(Object sender. EventArgs e) {}
        protected  void Application_End(Object sender. EventArgs e) {}
}
```

В некоторых отношениях класс Global действует как промежуточное звено между внешним клиентом и Web Form. Если вы работали с классическими ASP, то многие из этих событий будут вам знакомы. В общем можно сказать, что эти события позволяют нам реагировать на запуск и прекращение работы как web-приложения в целом, так и отдельных сеансов подключения.

Простой код ASP.NET на C#

Если мы сейчас обратимся по адресу нашего web-приложения, то среда выполнения ASP.NET вернет пустую страницу. Давайте исправим эту ситуацию и изме-

ним содержание файла default.aspx таким образом, чтобы нам возвращалась информация о произведенном запросе HTTP (свойство System.Web.UI.Page.Response нам предстоит вскоре рассмотреть более подробно):

```
<body MS_POSITIONING="GridLayout">
        <h1>
                <b>I am:</b>
        </h1>
        <%=this.ToString() %>
        <h1>
                <b>You are:</b>
        </h1>
        <%=Request.ServerVariables["HTTP_USER_AGENT"] %>

        <form method="post" runat="server" ID="Form1">
        </form>
</body>
```

Откомпилируем проект и запустим его на выполнение. Нам вернется страница HTML (рис. 14.33), с информацией о браузере, из которого был отправлен запрос, а также о сущности, которая этот запрос приняла (то есть просто имя страницы ASP.NET).

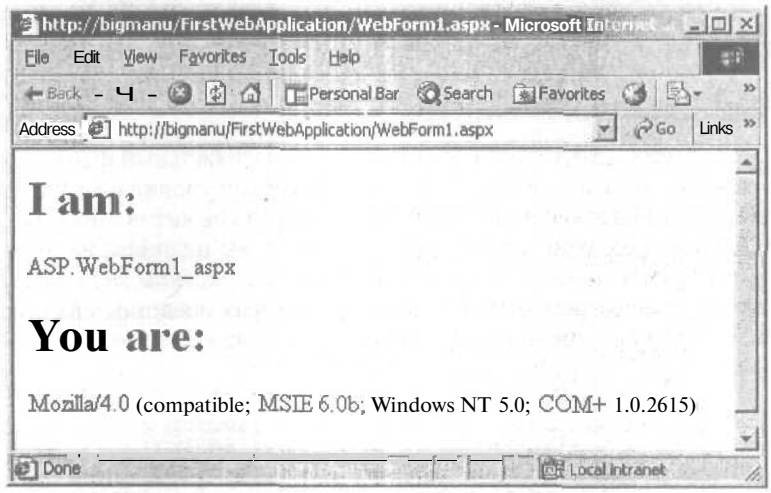


Рис. 14.33. Кто есть кто в ASP.NET

Все выглядит очень похоже на работу с классическими ASP. Однако есть и существенные отличия. Например, Request — это теперь свойство объекта, производного от Page. Кроме того, в тегах <%...%> теперь находится не код языка скриптов, а полноценный код C#:

```
<h1><b>I am: </b> <%=this.ToString() %></h1>
```

Архитектура web-приложения ASP.NET

Теперь, когда у нас уже есть опыт создания очень простого приложения ASP.NET, пора разобраться с основными особенностями архитектуры ASP.NET в целом. Как

мы помним, среда выполнения поместила в исходный скриптовый блок на нашей странице следующий блок с таинственным атрибутом Codebehind:

```
<%@ Page language="c#" Codebehind="defaul.aspx.cs"
AutoEventWireup="false" Inherits="FirstWebApplication.WebForm1" %>
```

Главное различие между классическими ASP и ASP.NET заключается в том, что страница *.aspx, которую запрашивает внешний клиент, представлена уникальным классом C#, на который и указывает атрибут Codebehind. Когда клиент запрашивает страницу, среда выполнения ASP.NET создает объект этого класса. Этот файл не показывается в окне Solution Explorer. Чтобы его открыть, необходимо щелкнуть правой кнопкой мыши на уже открытом файле *.aspx и в контекстном меню выбрать View Code. Изначально содержимое этого файла выглядит следующим образом:

```
namespace FirstWebApplication
{
    using System;
    using System.Collections;
    using System.ComponentModel;
    using System.Data;
    using System.Drawing;
    using System.Web;
    using System.SessionState;
    using System.Web.UI;
    using System.Web.UI.WebControls;
    using System.Web.UI.HtmlControls;

    public class WebForm1 : System.Web.UI.Page
    {
        public WebForm1()
        {
            Page.Init += new System.EventHandler(Page_Init);
        }
        protected void Page_Load(object sender, System.EventArgs e)
        {
            // Добавляем пользовательский код для инициализации страницы
        }
        protected void Page_Init(object sender, EventArgs e)
        {
            InitializeComponent();
        }

        private void InitializeComponent()
        {
            this.Load += new System.EventHandler(this.Page_Load);
        }
    }
}
```

Как мы видим, исходный код очень простой. Конструктор класса устанавливает обработчик события для события Init. Реализация этого обработчика вызывает метод InitializeComponentO, который, в свою очередь, устанавливает обработчик события Load.

Тип System.Web.UI.Page

Как мы видим, любая страница ASP.NET представлена классом, производным от System.Web.UI.Page. Этот класс определяет свойства, методы и события, общие для

всех страниц, предназначенных для выполнения в среде ASP.NET. Некоторые наиболее важные свойства этого класса представлены в табл. 14.4.

Таблица 14.4. Свойства класса Page

Свойство	Описание
Application	Возвращает объект HttpApplicationState
Cache	Возвращает объект Cache, в котором хранятся данные приложения, частью которого является эта страница
IsPostBack	Возвращает значение, определяющее, была ли эта страница загружена клиентом в первый раз или она загружена повторно в ответ на переданные клиентом данные
Request	Возвращает объект HttpRequest, используемый для получения информации о входящем запросе HTTP
Response	Возвращает объект HttpResponse, при помощи которого можно скомпоновать данные, возвращаемые браузеру клиента
Server	Возвращает объект HttpServerUtility
Session	Возвращает объект System.Web.SessionState.HttpSessionState, при помощи которого можно получить информации о текущем сеансе подключения

Как мы видим, свойства класса Page обеспечивают нам те же возможности, которые были в нашем распоряжении в классических ASP. В классе Page также имеется несколько унаследованных методов (они используются редко) и несколько очень важных событий, представленных в табл. 14.5.

Таблица 14.5. Наиболее важные события класса Page

Событие	Описание
Init	Это событие происходит, когда страница инициализируется. Это — первое событие жизненного цикла страницы
Load	Это событие происходит после события Init Обработчик этого события можно использовать для настройки любых элементов управления WebForm
Unload	Происходит при выгрузке объекта из памяти. Можно использовать, к примеру, для освобождения ресурсов

Обработчик события Load — лучшее место установления соединения с источником данных (например, для заполнения на форме элемента управления WebForm DataGrid) или выполнения других подготовительных действий. Обработчик события Unload можно использовать, например, для закрытия этого соединения и выполнения других аналогичных действий.

Связка *.aspx/Codebehind

Помимо готовых членов, унаследованных от Page, мы можем определить в своем классе C# собственные члены, которые могут быть вызваны (не напрямую) при помощи блоков <%...%> в файле *.aspx. В классических ASP все дополнительные возможности приходилось определять непосредственно в коде файла *.asp. В результате этот файл превращался в совершеннейшие джунгли из тегов HTML и кода VBScript (или JavaScript). Читать такие файлы было очень тяжело, а использовать их код повторно — еще тяжелее.

В ASP.NET эта проблема решена за счет того, что код представления (то есть код для генерации кода HTML) помещен в файл *.aspx, а прочая программная логика — обычным образом в файл C# *.aspx.cs.

Как же происходит обращение к пользовательским членам класса, производного от Page? Предположим, что мы определили в таком классе простенькую функцию, возвращающую текущее значение даты и времени:

```
public class WebForm1 : System.Web.UI.Page
{
        // Автоматически сгенерированный код...

        public string GetDateTime()
        {
                return DateTime.Now.ToString();
        }
}
```

Обратиться к этой функции из файла *.aspx можно так:

```
<body>
        <!- Получаем информацию о времени от класса C# ->
        <% Response.Write(GetDateTime()); %>
...
        <form method="post" runat="server" ID=Form1>
        </form>
</body>
```

Конечно, можно использовать нужные нам унаследованные от Page члены непосредственно внутри класса C#. Например, можно определить нашу функцию так:

```
public class WebForm1 : System.Web.UI.Page
{
        // Автоматически сгенерированный код...

        public void GetDateTime()
        {
                Response.Write("It is now " + DateTime.Now.ToString());
        }
}
```

А затем просто вызывать эту функцию:

```
<!- Получаем время ->
<% GetDateTime(); %>
```

Свойство Page.Request

Множество web-приложений построено по одному и тому же принципу: клиент заходит на web-сайт, заполняет форму HTML своей пользовательской информацией и нажимает на кнопку Submit этой формы, чтобы передать информацию на сервер. В большинстве случаев в теге форм используются атрибуты action и method. Первый определяет адрес получателя информации на сервере, а второй — метод передачи информации:

```
<form name=MainForm action="http://localhost/default.asp" method=get ID=Form1>
```

В ASP.NET свойство Page.Request позволяет получить доступ к данным, отправленным пользователем в виде запроса HTTP. Если разобраться, что делает это свойство, то выяснится, что оно взаимодействует с объектом класса HttpRequest. Неко-

торые наиболее важные свойства этого класса представлены в табл. 14.6. Если вы работали с классическими ASP, то они покажутся вам очень знакомыми.

Таблица 14.6. Свойство типа HttpRequest

Свойство	Описание
ApplicationPath	Возвращает виртуальный путь к приложению, выполняющемуся на сервере
Browser	Позволяет получить информацию о возможностях браузера клиента
ContentType	Определяет тип содержимого MIME для входящего запроса. Это свойство доступно только для чтения
Cookies	Возвращает набор клиентских cookie
FilePath	Возвращает виртуальный путь к запрашиваемому файлу. Это свойство доступно только для чтения
Files	Возвращает набор файлов, загруженных клиентов (формат MIME для файлов из нескольких частей)
Filter	Позволяет получить или установить фильтр, используемый для чтения потока входящих данных
Form	Позволяет получить набор переменных Form
Headers	Позволяет получить набор заголовков HTTP
HttpMethod	Определяет метод передачи данных, используемый клиентом (GET, POST). Это свойство доступно только для чтения
IsSecureConnection	Позволяет получить информацию о том, является ли установленное соединение защищенным (с применением SSL). Это свойство доступно только для чтения
Params	Возвращает комбинированный набор QueryString + Form + ServerVariable + + Cookie
QueryString	Возвращает набор переменных QueryString
RawUrl	Возвращает текущий запрос клиента в виде адреса URL
Requesttype	Определяет метод передачи данных, используемых клиентом (GET, POST)
ServerVariables	Возвращает набор переменных web-сервера
UserHostAddress	Возвращает IP-адрес компьютера удаленного клиента
UserHostName	Возвращает имя DNS компьютера удаленного клиента

При помощи этих свойств можно получить любую возможную информацию о запросе пользователя. Мы уже использовали этот объект ранее, не очень об этом догадываясь. Например, когда мы получали информацию о браузере пользователя при помощи строки следующего вида:

```
<b>You Are: </b><%= Request.ServerVariables[HTTP_USER_AGENT"] %>
```

реально мы обращались к свойству возвращаемого объекта HttpRequest:

```
<b>You Are: </b>
<%
        HttpRequest r;
        r = this.Request;
        Response.Write(r.ServerVariables["HTTP_USER_AGENT"]);
%>
```

Как получать информацию о передаваемых пользователем данных привычными средствами C#, мы уже разобрались. Однако нам нужно как-то на них отреаги-

решать. И в этом нам поможет свойство Page. Response и соответствующий ему класс HttpResponse.

Свойство Page. Response

Свойство Page. Response возвращает объект класса HttpResponse. В этом классе предусмотрено множество свойств, которые предназначены для одной цели — помочь нам скомпоновать ответ в виде кода HTML (то есть web-страницу), который будет возвращен браузеру клиента. Некоторые наиболее важные свойства этого класса представлены в табл. 14.7.

Таблица 14.7. Свойства класса HttpResponse

Свойство	Описание
Cache	Возвращает информацию о кэшировании для web-страницы (время устаревания и т. п.)
ContentEncoding	Позволяет определить кодировку для возвращаемых клиенту данных
ContentType	Позволяет определить тип MIME для возвращаемых клиенту данных
Cookies	Возвращает коллекцию HttpCookie, отправленных в текущем запросе
Filter	Определяет объект фильтра, который может быть использован для внесения изменений в данные HTTP перед отправкой их клиенту
IsClientConnected	Позволяет получить информацию о том, подключен ли клиент к серверу
Output	Используется для добавления пользовательских данных в возвращаемый клиенту ответ на запрос
OutputStream	То же самое, но для добавления двоичных данных
StatusCode	Позволяет определить код состояния HTTP для переданных клиенту данных
StatusDescription	Позволяет получить строку состояния HTTP для переданных клиенту данных
SupressContent	Позволяет получить или установить значение, определяющее, будут ли данные отправлены клиенту

Кроме того, в классе HttpResponse определены важные методы, представленные в табл. 14.8.

Таблица 14.8. Методы класса HttpResponse

Метод	Описание
AppendHeader()	Добавляет заголовок HTTP в возвращаемые клиенту данные
AppendToLog()	Добавляет пользовательскую информацию в файл журнала IIS
Clear()	Очищает все заголовки и все содержимое буфера для возвращаемых данных
Close()	Закрывает соединение с клиентом
End()	Отправляет все содержимое буфера для возвращаемых данных клиенту, а после этого закрывает соединение
Flush()	Отправляет все содержимое буфера для возвращаемых данных клиенту
Redirect()	Перенаправляет клиента по указанному адресу URL
Write()	Добавляет значение в данные, возвращаемые клиенту
WriteFile()	Метод многократно перегружен. Используется для направления файла напрямую браузеру клиента

Наиболее важный и часто используемый метод класса HttpResponse — это метод Write(), который позволяет добавлять значения в набор возвращаемых клиенту данных. Этот метод можно вызывать как явно:

```
<b>You are: </b>
<%
        HttpRequest r;
        r = this.Request;

        HttpResponse rs;
        rs = this.Response;

        rs.Write(r.ServerVariables["HTTP_USER_AGENT"]);
%>
```

так и косвенно, в стиле классических ASP:

```
<%= Request.ServerVariables["HTTP_USER_AGENT"] %>
```

И в том и в другом случае результат будет совершенно одинаковым.

Свойство Page.Application

Свойство Application класса Page обеспечивает доступ к объекту класса HttpAppl IcationState. Как уже говорилось выше, HttpAppl icationState предоставляет разработчикам возможность управления информацией, общей для всех сеансов подключения к приложению ASP.NET. Некоторые наиболее важные свойства HttpAppl icationState представлены в табл. 14.9.

Таблица 14.9. Свойства типа HttpApplicationState

Свойство	Описание
AllKeys	Возвращает набор всех объектов, относящихся к состоянию приложения
Count	Позволяет получить количество объектов в наборе, относящихся к состоянию приложения
Keys	Возвращает объект NameObjectCollectionBase.KeysCollection, используемый для хранения всех ключей состояния приложения объекта NameObjectCollectionBase
StaticObjects	Позволяет получить доступ ко всем объектам, объявленным в теге <x runat=server> </x> в файле приложения ASP.NET

При создании переменной, которая будет доступна из всех сеансов подключения, необходимо использовать пару имя — значение (например, firstUser= "chuck"). а затем добавить ее во внутреннюю коллекцию KeysCollection. Для этого можно использовать индексатор класса:

```
public class WebForm1 : System.Web.UI.Page
{
        protected void Page_Load(object sender. EventArgs e)
        {
                if (!IsPostBack)
                {
                        // Создаем переменную уровня всего приложения
                        Application["AppString"] = "Initial App Value";
```

```
        }
    }
...
}
```

Если нам затем потребуется обратиться к этому значению, просто извлечем его при помощи того же свойства:

```
string appVar = "App: " + Application["AppString"];
```

Код приложений WebForm1.aspx и WebForml.aspx.cs можно найти в подкаталоге Chapter 14.

Отладка и трассировка приложений ASP.NET

Если вам приходилось работать с Visual InterDev, то вы, наверное, помните, что отлаживать приложения в нем было не очень удобно. ASP.NET выигрывает и в этом отношении: при создании проектов ASP.NET мы можем использовать те же самые средства отладки, что и для любых других проектов Visual Studio.NET. Например, мы можем устанавливать брейкпойнты в скриптовых блоках (или в файлах C#), запускать сеансы отладки, производить пошаговое выполнение и т. п. (рис. 14.34).

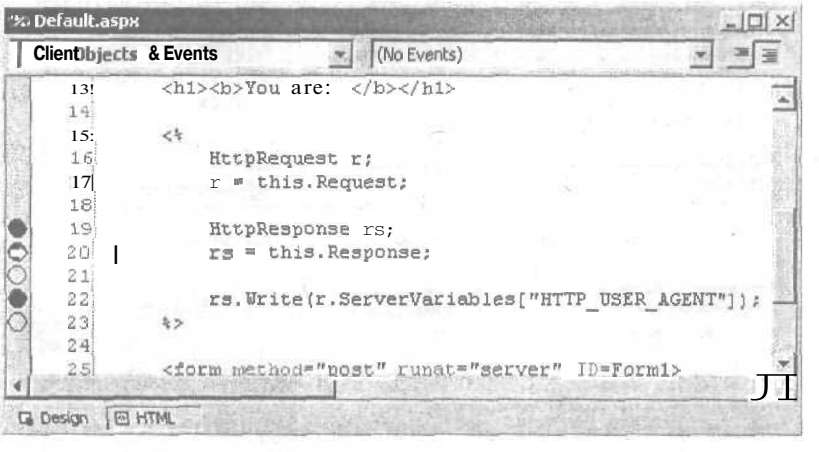


Рис. 14.34. Установка брейкпойнтов

Кроме того, мы можем осуществлять трассировку файлов *.aspx просто путем добавления атрибута trace в открывающий скриптовый блок:

```
<%@ Page Language="c#" Codebehind="WebForm1.aspx.cs"
AutoEventWireup="False" Inherits="FirstWebApplication.WebForm1" trace="true"
```

В результате в конец возвращаемого клиенту файла HTML будет добавлена информация трассировки (рис. 14.35).

Мы можем добавлять и свои собственные данные трассировки. Для этого используется класс Trace. Каждый раз, когда нам нужно добавить свое собственное сообщение трассировки, мы просто используем метод Write этого класса:

```
<%
    Trace.Write("App Ctegory", "About to determine agent...");
    HttpRequest r;
    r - this.Request;

    HttpResponse rs;
    rs = this.Response;

    rs.Write(r.ServerVariables["HTTP_USER_AGENT"]);
%>
```

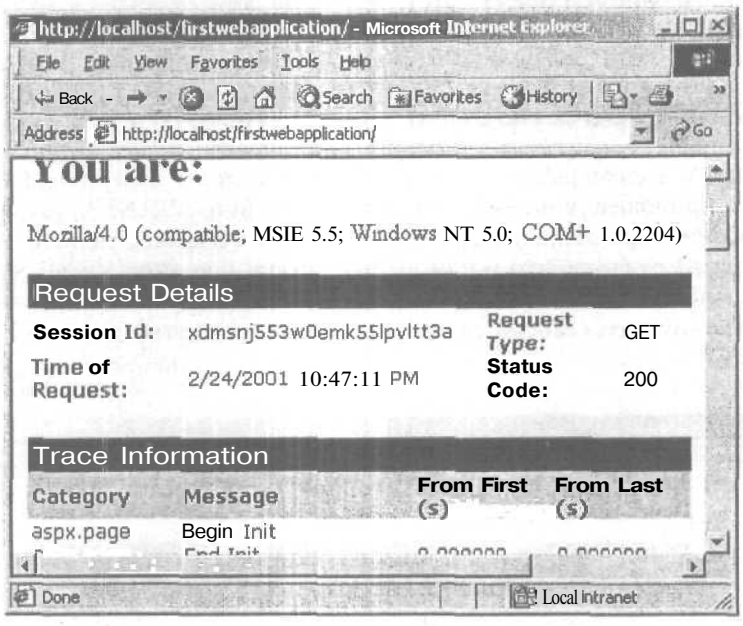


Рис. 14.35. Получение информации трассировки

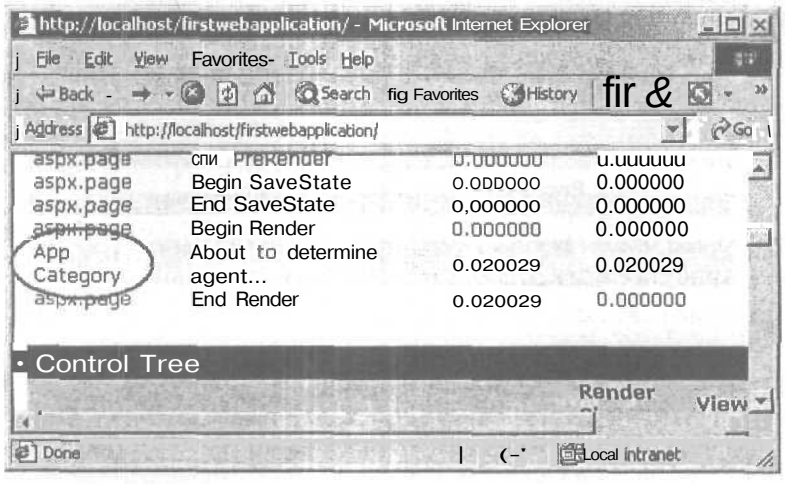


Рис. 14.36. Пользовательское сообщение трассировки

Наше пользовательское сообщение трассировки представлено на рис. 14.36,

Мы уже достаточно много узнали про запросы HTTP и создание кода HTML в ответ на эти запросы. Однако мы еще ничего не сказали про элементы графического интерфейса, которые можно использовать как для приема данных от пользователя, так и для возврата ему информации в ответ на его запрос, хотя здесь ASP.NET предлагает нам еще большее количество новых и очень привлекательных возможностей.

Элементы управления WebForm

Одно из важнейших достоинств ASP.NET заключается в том, что при его использовании резко упрощается создание элементов пользовательского интерфейса на web-страницах. Элементы управления WebForm (их также называют серверными элементами управления (server controls) или элементами управления Web (Web controls)) определены в пространстве имен System.Web.UI.WebControls. Их основное назначение — избавить нас от трудоемкой работы по созданию элементов управления HTML на web-странице вручную.

Вот пример. Если мы создаем классическую страницу ASP и у нас есть необходимость разместить на ней несколько текстовых полей, то нам придется писать теги для каждого текстового поля на странице вручную. В ASP.NET нам достаточно будет перетащить на шаблон времени разработки графические элементы управления из Toolbox, а затем их настроить. Например, для текстового поля будет автоматически сгенерирован следующий код:

```
<form method="post" runat="server">
      <asp:TextBox id=TextBox1 style="Z-INDEX: 101; LEFT; 27px; POSITION: absolute;
                                                          TUP: 30px" runat="server">
      </asp:Text8ox>
</form>
```

Когда придет время отвечать на запрос пользователя, среда выполнения ASP.NET автоматически преобразует этот код в тег HTML следующего вида:

```
<input name="textBox1" type="text" id="TextBox1" style="Z-INDEX: 101; LEFT: 27px;
POSITION: absolute: TOP; 3Opx" />
```

Ну и что в этом хорошего, спросите вы. Не проще ли вообще написать код для текстового поля вручную? Все будет проще, и в итоге окажется меньше кода на странице ASP. Однако не все так просто. Во-первых, при использовании элементов управления WebForm нам может вообще не потребоваться писать вручную код HTML. А во-вторых, не все элементы управления такие простые, как текстовые поля. Например, если нам нужно поместить на генерируемую web-страницу большую таблицу, или баннерную рулетку, или календарный элемент управления, или элемент управления DataGrid, то применение элементов управления WebForm может сэкономить вам многие часы работы.

Еще одно преимущество элементов управления WebForm заключается в том, что с ними очень удобно работать с программной точки зрения: каждому из этих элементов управления соответствует класс в библиотеке базовых классов .NET, и *мы* можем работать с ними, как с любыми другими классами, и в файле *.aspx, и в производном от Page классе C#. Кроме того, в любом элементе управления WebForm также определен набор событий, которые будут обрабатываться *на сервере* (более подробно об этом будет рассказано чуть ниже).

И последнее, о чем тоже стоит упомянуть. При использовании элементов управления WebForm в нашем распоряжении появляется целый набор возможностей для проверки ввода данных пользователем. Таким образом, мы избавляемся не только от необходимости вручную писать теги для элементов управления HTML, но и от ручного создания клиентских скриптов JavaScript для проверки вводимых данных. (Конечно, если нам все же нужно использовать код JavaScript на нашей странице, никто нам этого делать не запрещает.)

Создание элементов управления WebForm

При создании проекта на основе шаблона Web Application в нашем распоряжении всегда есть набор элементов управления WebForm. Для того чтобы ими воспользоваться, достаточно перейти на соответствующую вкладку в Toolbox (рис. 14.37).

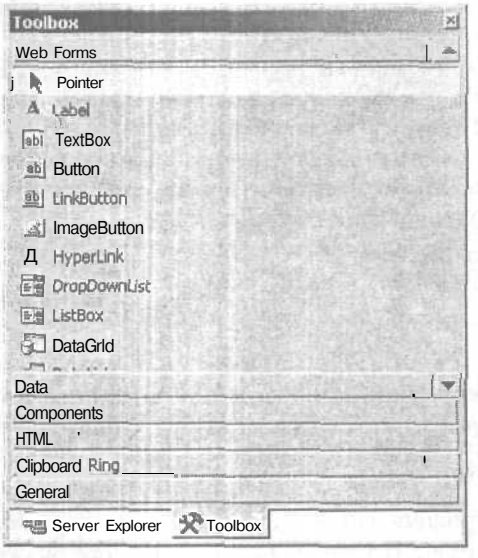


Рис. 14.37. Набор элементов управления WebForm

Настройку параметров каждого из элементов управления WebForm можно производить при помощи обычного окна свойств в IDE Visual Studio.NET. Эти элементы управления очень похожи на элементы управления Windows Forms, и поэтому проблем с настройкой подавляющего большинства свойств у нас не будет. Например, для текстового поля в нашем распоряжении набор свойств, представленный на рис. 14.38.

При внесении изменений через окно свойств для элемента управления все изменения сразу же записываются напрямую в файл *.aspx. Например, если для нашего текстового поля txtEMail мы изменим значения свойств BorderStyle, BorderWidth, BackColor, BorderColor и ToolTip, в теге <asp:textbox> появятся новые пары имя — значение:

```
<asp:textbox id=txtEMail runat="server" BorderStyle="Ridge" BorderWidth="5px"
BackColor="PaleGreen" BorderColor="DarkOliveGreen" ToolTip="Enter your e-mail here...">
</asp:TextBox>
```

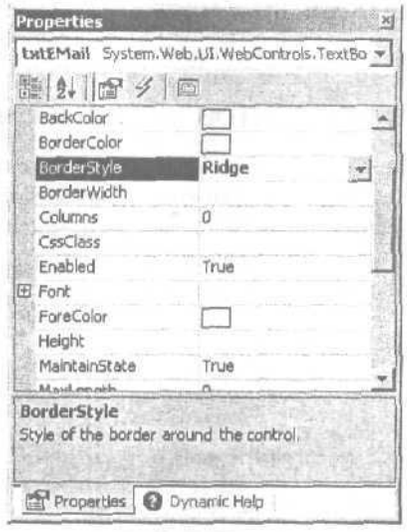


Рис. 14.38. Набор свойств для текстового поля

А вот как выглядит итоговый результат в виде кода HTML, который вернется пользователю:

```
<input name="txtEMail" type ="text" value="fdfdf" id="txtEMail" title="Enter your e-mail
here..." style="background-color:PaleGreen;border-color:DarkOliveGreen;border-
width:5px;border-style:Ridge;" />
```

Давайте разберемся с синтаксисом записей для элементов управления WebForm в файле *,aspx. Для каждого элемента управления используется синтаксис, очень напоминающий формат XML. Открывающим тегом всегда будет <asp: тип_элемента_управления runat="server">, а закрывающим — </asp: тип_элемента_управления>. Вот два примера таких тегов:

```
<asp:TextBox id=TextBox1 style="Z-INDEX: 101; LEFT: 27px; POSITION: absolute; TOP: 30px"
runat="server"> </asp:TextBox>
<asp:Button id=Button1 style="Z-INDEX: 102; LEFT: 26px; POSITION: absolute; TOP: 66px"
runat="server" DESIGNTIMEDRAGDROP="21" Text="Button"> </asp:Button>
```

Атрибут runat="server" означает, что это — элемент управления WebForm, предназначенный для выполнения на сервере. Прежде чем его код отправится к клиенту, он будет преобразован средой выполнения ASP.NET в привычный код HTML.

В файле C#, который указан в атрибуте Codebehind, код также изменился. В нем появились новые объекты, представляющие элементы управления. Имена этих объектов совпадают с идентификаторами элементов в тегах файла *.aspx:

```
public class WebForm1 : System.Web.UI.Page
{
        protected System.Web.UI.WebControls.Button btnSubmit;
        protected System.Web.UI.WebControls.CheckBox ckBoxNewsLetter;
        protected System.Web.UI.WebControls.TextBox txtEMail;
        protected System.Web.UI.WebControls.TextBox txtLName;
        protected System.Web.UI.WebControls.TextBox txtFName;
...
}
```

Конечно же, мы можем программным образом работать с этими объектами привычными способами C#.

Иерархия классов элементов управления WebForm

Все элементы управления WebForm — это классы, производные от базового класса System.Web.UI.WebControls.WebControl. Для этого класса, в свою очередь, базовым является System.Web.UI.WebControls.Control, а тот уже происходит непосредственно от System.Object. Например, иерархия классов для элемента управления Button выглядит так, как показано на рис. 14.39.

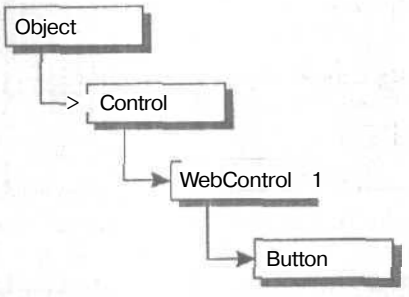


Рис. 14.39. Иерархия классов для элемента управления Button

И класс Control, и класс WebControl наделяют производные классы каждый своим набором очень важных членов. Свойства, определенные в классе Control, представлены в табл. 14.10.

Таблица 14.10. Свойства базового класса Control

Свойство	Описание
ID	Позволяет получить или установить идентификатор элемента управления
MaintainState	Позволяет работать с состоянием просмотра для элемента управления (более подробно об этом будет рассказано ниже)
Page	Возвращает объект Page, то есть страницу, на которой находится данный элемент управления
Visible	Позволяет получить или определить видимость элемента управления (то есть будет он выводиться на странице или нет)

Свойства, определенные в классе WebControl, в основном позволяют определять внешний вид элемента управления (табл. 14.11).

Таблица 14.11. Свойства базового класса WebControl

Свойство	Описание
BackColor	Позволяет получить или установить цвет фона элемента управления
BorderColor	Позволяет получить или установить цвет рамки вокруг элемента управления
BorderStyle	Позволяет получить или установить стиль рамки вокруг элемента управления
BorderWidth	Позволяет получить или установить толщину рамки вокруг элемента управления

Свойство	Описание
Enabled	Позволяет получить или установить значение, определяющее доступность элемента управления
Font	Позволяет получить информацию об используемом шрифте
ForeColor	Позволяет получить или установить цвет переднего плана (обычно это — цвет надписи) для элемента управления
Height Width	Определяет высоту и ширину элемента управления
TabIndex	Позволяет получить или установить значение TabIndex (очередности перехода по клавише Tab)
ToolTip	Позволяет получить или установить значение ToolTip — всплывающей подсказки, появляющейся при наведении на элемент управления указателя мыши

Виды элементов управления WebForm

Все множество элементов управления WebForm можно поделить на четыре основные разновидности:

- базовые элементы управления;
- элементы управления с дополнительными возможностями;
- элементы управления для работы с источниками данных;
- элементы управления для проверки вводимых пользователем данных.

Если вы усвоили материал главы 10, в которой рассказывалось об элементах управления Windows Forms, вы будете чувствовать себя как дома. Единственное, о чем постоянно нужно помнить — это о самом принципиальном различии между элементами управления Windows Forms и WebForm. Первые в итоге преобразуются в набор вызовов Win32 API, а вторые — в набор тегов HTML.

Базовые элементы управления WebForm

К базовым элементам управления WebForm относятся те из них, которым есть прямые соответствия в HTML. Например, для отображения списка моделей автомобилей, представленного на рис. 14.40, можно использовать элемент управления ListBox с прилагающимися к нему элементами управления ListItem:

```
<asp:ListBox id=ListBox1 runat="server" Width="86" Height="69">
        <asp:ListItem value="BMW">BMW</asp:ListItem>
        <asp:ListItem value="Jetta">Jetta</asp:ListItem>
        <asp:ListItem value="Colt">Colt</asp:ListItem>
        <asp:ListItem value="Grand Am">Grand Am</asp:ListItem>
</asp:ListBox>
```

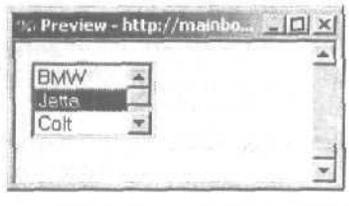


Рис. 14,40. Элемент управления ListBox

Перед тем как отправиться к клиенту, этот код будет преобразован средой выполнения ASP.NET в следующий набор тегов HTML:

```
<select name="ListBox1" id="ListBox1" size="5" style="height:69px;width:86px;">
        <option value="BMW">BMW</option>
        <option value="Jetta">Jetta</option>
        <option value="Colt">Colt</option>
        <option value="Grand Am">Grand Am</option>
. </select>
```

Некоторые базовые элементы управления WebForm представлены в табл. 14.12.

Таблица 14.12. Базовые элементы управления WebForm

Элемент управления	Описание
Button ImageButton	Разновидности кнопок
CheckBox CheckBoxList	Флажок (CheckBox) или окно списка с несколькими флажками (CheckBoxList)
DropDownList ListBox ListItem	Эти типы предназначены для создания различных разновидностей списков
Image Panel Label	Эти типы представляют контейнеры для статического текста и изображений (а также средство для их группировки)
RadioButton RadioButtonList	Стандартный переключатель (RadioButton) или окно списка с набором переключателей (RadioButtonList)
TextBox	Текстовое окно для ввода данных пользователем. Может быть настроено для приема одной строки текста или нескольких строк

Работа с базовыми элементами управления WebForm очень похожа на работу с их аналогами в Windows Forms. Мы не будем рассматривать все элементы управления WebForm подряд, а остановимся на нескольких стандартных ситуациях.

Группа переключателей

Переключатели обычно объединяются в группы. В одной группе одновременно может быть выбран только один переключатель. Например, если нам необходимо создать пользовательский интерфейс, представленный на рис. 14.41, код может быть таким:

```
<body>
<p><font size=5><em>How shall we contact you?</em></font></p>

<p><asp:RadioButton id=RadioHome runat="server" Text="Contact me at home"
                                                GroupName="ContactGroup">
</asp:RadioButton></p>

<p><asp:RadioButton id=RadioWork runat="server" Text="Contact me at work"
                                                GroupName="ContactGroup">
</asp:RadioButton></p>

<p><asp:RadioButton id=RadioDontBother runat="server" Text="Don't bother me..."
                                                GroupName="ContactGroup">
</asp:RadioButton></p>
</body>
```

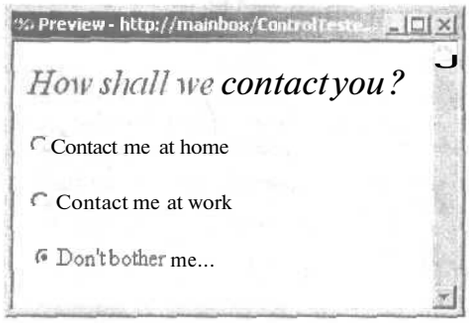


Рис. 14.41. Группа переключателей на web-странице

Рис. 14.42. Текстовое поле для ввода нескольких строк

Обратите внимание, что для каждого объекта RadioButton предусмотрен атрибут GroupName. Если значение этого атрибута у нескольких переключателей одно и то же (как в нашем случае), одновременно может быть выбран только один из них.

Текстовое поле для ввода нескольких строк с полосой прокрутки

Еще одним часто используемым элементом управления является текстовое поле для ввода нескольких строк (рис. 14.42).

Как вы, наверное, уже догадываетесь, чтобы в поле можно было вводить несколько строк, необходимо установить значение соответствующего атрибута. Выглядеть это может так:

```
<p><aps:TextBox id=TextBox1 reunat="server" Width="183" Height="96" TextMode="MultiLine"
BorderStyle="Ridge"> </asp:TextBox></p>
```

При установке для атрибута TextMode значения Multi Line у текстового поля автоматически возникает полоса прокрутки (когда введенные в него данные уже не могут поместиться в отображаемой области).

Элементы управления с дополнительными возможностями

К этой группе относятся элементы управления, для которых не предусмотрено прямых аналогов в HTML. Их всего два, и они представлены в табл. 14.13.

Таблица 14.13. Элементы управления WebForm с дополнительными возможностями

Элемент управления	Описание
AdRotator	Баннерная рулетка: набор из нескольких пар изображение — альтернативный текст, которые сменяют друг друга. Для настройки используется специальный код в формате XML
Calendar	Этот элемент управления возвращает код HTML, представляющий календарь

Элемент управления Calendar

Для элемента управления Calendar не существует прямого эквивалента в HTML. Однако необходимость в помещении на web-страницу календаря возникает очень часто. Поэтому в ASP.NET был предусмотрен специальный элемент управления, который преобразуется средой выполнения ASP.NET в набор тегов HTML, представляющий календарь. Например, предположим, что мы разместили на своей web-странице этот элемент управления при помощи следующего кода:

```
<asp:Calendar id=Calendar1 runat="server"></asp:Calendar></p>
```

Вы удивитесь, увидев, какое количество кода HTML сгенерировала среда выполнения ASP.NET, встретив на странице такую строку! Кода так много, что мы даже не будем его здесь приводить. Лучше посмотрите его сами: поместите объект Calendar из Toolbox на графический шаблон времени разработки, сохраните файл *.aspx и обратитесь к нему из web-браузера. После этого щелкните на открывшейся в окне браузера странице правой кнопкой мыши и в контекстном меню выберите команду View Code. Откроется окно Notepad, в котором вы сможете найти код для элемента управления Calendar (рис. 14.43).

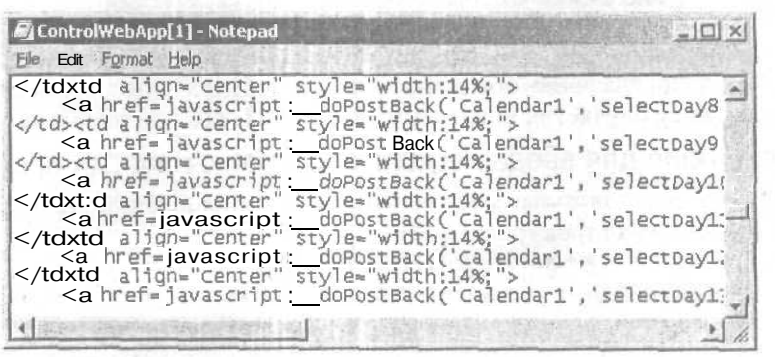


Рис. 14.43. Для элемента управления Calendar генерируется весьма объемный код HTML

В элементе управления Calendar предусмотрена масса возможностей для настройки. Одно из свойств, которое может представлять интерес, — это свойство SelectionMode. По умолчанию в календаре можно выбирать только один день (что соответствует значению по умолчанию SelectionMode - Day). Однако мы можем воспользоваться и другими допустимыми значениями этого свойства:

* None — вообще ничего нельзя будет выбирать, то есть календарь будет предназначен исключительно для справочных целей;

- DayWeek — можно выбирать один день или целую неделю;
- DayWeekMonth — можно будет выбрать день, неделю или месяц.

Например, если мы установим значение DayWeekMonth, в возвращаемом коде HTML будет предусмотрен дополнительный столбец слева (для выбора недели целиком) и флажок в верхнем левом углу (для выбора всего месяца). Вот код, использующий все возможные свойства этого элемента управления (не пугайтесь, все можно установить через окно свойств в Visual Studio.NET):

```
<asp:Calendar id=Calendar1 runat="server" SelectionMode="DayWeekMonth"
DayNameFormat="FirstLetter" BackColor="White"
SelectionStyle-ForeColor="#336666"
SelectorStyle-BackColor="#CCFF99"
TodayDayStyle-BackColor="#99CCCC" DayHeaderStyle-Height="1px"
DayHeaderStyle-ForeColor="#336666"
DayHeaderStyle-BackColor="#99CCCC
Font-Size="8pt" Font-Names="Verdana" Height="200"
OtherMonthDayStyle-ForeColor="#999999"
TitleStyle-Font-Style="11pt"
TitleStyle-Font-Bold="True" TitleStyle-ForeColor="#CCFF99"
TitleStyle-BackColor="#003399" ForeColor="#003399" BorderColor="#3366CC" Width="221"
SelectedDayStyle-ForeColor="#CCFF99"
SelectedDayStyle-BackColor="#009999"
TodayDayStyle-ForeColor="White" BorderWidth="1px"
TitleStyle-BorderStyle="Solid" TitleStyle-BorderWidth="1px"
TitleStyle-BorderColor="#3366CC" WeekendDayStyle-BackColor="#CCCCFF"
SelectedDayStyle-Font-Bold="True" CellPadding="1">
</asp:Calendar>
```

То, как теперь должен выглядеть наш календарь в окне Internet Explorer, показано на рис. 14.44.

Рис. 14,44. Элемент управления Calendar в окне браузера клиента

Элемент управления AdRotator (баннерная рулетка)

Элемент управления AdRotator (баннериая рулетка) был и в классических ASP, однако в ASP.NET он был дополнен новыми возможностями. Задача этого элемента управления проста и понятна: менять картинки в окне браузера через заданные промежутки времени. Обычно, конечно, с его помощью гоняют рекламу. При размещении этого элемента управления на шаблоне страницы времени разработки на ней будет лишь помечено место, где будут находиться баннеры. Все остальное придется делать вручную. Если точнее, то придется указать для свойства AdvertisementFile имя файла в формате XML, в котором будут храниться настройки баннерной рулетки, а затем написать этот самый файл.

Формат Advertisement File очень прост. Для каждого баннера создается отдельный тег <Ad> (от advertisement). Как минимум, в этом теге должен быть указан путь к файлу изображения (ImageUrl), то есть баннера, адрес URL, на который клиент перейдет при щелчке на этом баннере (TargetUrl), альтернативный текст (AlternativeText), который будет периодически сменять изображение или появляться при наведении на него указателя мыши, и вес этого баннера в общем времени показа (Impressions). Например, мы можем создать следующий файл в формате xml (пусть он называется ads.xml):

```
<Advertisements>
    <Ad>
        <ImageUrl>SlugBug.jpg</ImageUrl>
        <TargetUrl>http://www.Cars.com</TargetUrl>
        <AlternateText>Your new Car?</AlternateText>
        <Impressions>80</Impressions>
    </Ad>

    <Ad>
        <ImageUrl>car.gif</ImageUrl>
        <TargetUrl>http://www.CarSuperSite.com</TargetUrl>
        <AlternateText>Like this Car?</AlternateText>
        <Impressions>80</Impressions>
    </Ad>
</Advertisements>
```

После этого нам осталось убедиться, что файл XML и файлы изображений помещены в один виртуальный каталог с нашей страницей *.aspx и настроить для элемента управления AdRotator значение атрибута AdvertisementFile, например, так:

```
<asp:AdRotator id=AdRotator1 runat="server" Width="470"
Height="60" AdvertisementFile="ads.xml">
</asp:AdRotator>
```

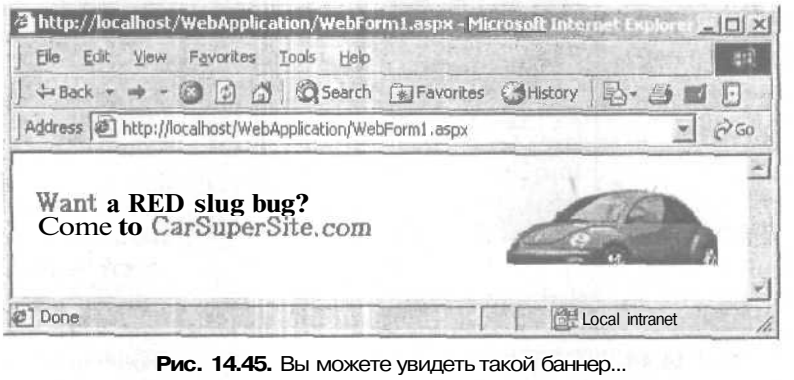


Рис. 14.45. Вы можете увидеть такой баннер...

Рис. 14.46. ...и такой

Разные «повороты» созданной нами баннерной рулетки представлены на рис. 14.45 и 14.46.

Свойства Height и Width элемента управления AdRotator определяют высоту и ширину баннера. Если наше изображение не будет подходить под эти размеры, оно будет растянуто или сжато.

Код приложения Controls можно найти в подкаталоге Chapter 14.

Элементы управления для работы с источниками данных

В ASP.NET предусмотрено два элемента управления WebForm, предназначенных для отображения данных, полученных из источника (обычно в качестве источника в приложениях ASP.NET выступает объект ADO.NET DataSet, который, в свою очередь, может быть, например, заполнен данными с сервера баз данных). Эти элементы управления представлены в табл. 14.14.

Таблица 14.14. Элементы управления WebForm, предназначенные для работы с источниками данных

Элемент управления	Описание
DataGrid	Элемент управления, который отображает содержимое объекта ADO.NET DataSet в виде таблицы
DataList	Элемент управления для выбора значений, заполняемый из источника данных

Кроме того, для работы с источниками данных можно настроить некоторые из базовых типов данных.

Элемент управления DataGrid

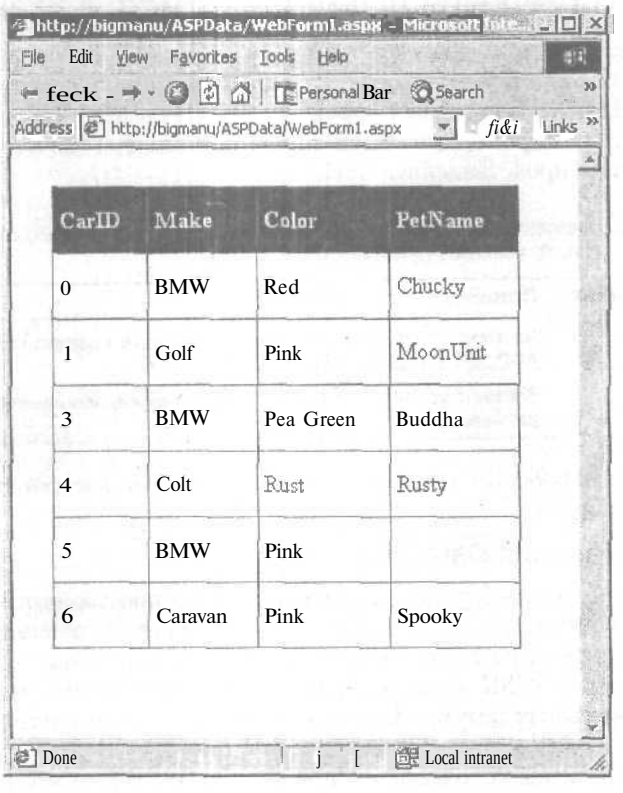
Одна из наиболее часто встречающихся задач в web-приложении — найти какие-то данные в источнике данных по запросу пользователя и вернуть их в табличном формате. В классических ASP это делалось путем создания объекта ADO Recordset и создания таблицы HTML «на лету» с использованием данных из этого объекта Recordset. Тех же самых результатов гораздо проще можно достичь при помощи элемента управления WebForm - DataGrid.

Рассмотрим применение DataGrid на примере. Предположим, что нам необходимо предоставить пользователю в ответ на его запрос данные из базы данных Cars (той са-

мой, с которой мы работали в главе 13). Первое, что нам нужно сделать — создать обработчик для события Load нашей страницы. В нем мы установим соединение с базой данных, создадим и заполним объект DataSet и укажем его в качестве источника данных для элемента управления DataGrid. Соответствующий код C# может выглядеть так:

```
using System.Data.SQL;
...
protected void Page_Load(object sender. EventArgs e)
{
        if(!IsPostBack)
        {
                // Помещаем в DataGrid данные из таблицы Inventory
                SqlConnection sqlConn = new SqlConnection();
                sqlConn.ConnectionString = "data source=.; initial catalog=Cars;
                                                integrated security=sspi;";
                SqlDataAdapter dsc = new SqlDataAdapter("Select * from Inventory",
                                                        sqlConn);
                DataSet ds = new DataSet();
                dsc.Fill(ds, "Inventory");

                DataGrid1.DataSource = ds.Tables["Inventory"].DefaultView;
                DataGridl.OataBindO:
        }
}
```

Результат представлен на рис, 14.47.



CarID	Make	Color	PetName
0	BMW	Red	Chucky
1	Golf	Pink	MoonUnit
3	BMW	Pea Green	Buddha
4	Colt	Rust	Rusty
5	BMW	Pink	
6	Caravan	Pink	Spooky

Рис. 14.47. Элемент управления DataGrid с данными, полученными из SQL Server

Еще немного об источниках данных

Как мы только что убедились, выводить содержимое объекта DataTable при помощи элементов управления WebForm (например, DataGrid) можно легко и просто. Однако достаточно часто возникает необходимость выводить на web-странице данные, которые хранятся другими способами. И в элементах управления WebForm предусмотрена возможность делать это, то есть выводить данные, которые находятся в каком угодно виде.

Например, предположим, что мы столкнулись со следующей ситуацией: нам необходимо заполнить элемент управления ListBox данными, которые в настоящее время хранятся в обычном строковом массиве (такая потребность возникает очень часто). Заполнение ListBox данными из этого массива производится точно так же, как заполнение данными DataGrid из объекта DataTable в предыдущем примере. Все удивительно просто:

```
protected void Page_load(object sender. EventArgs e)
{
        if(!IsPostBack)
        {
                // Создаем массив строковых значений - он нам нужен
                // для нашей демонстрации
                string[] carPetNames = { "Viper". "Hank". "Ottis", "Alponzo", "Cage", "TB" };
                // petNameList - это наш элемент управления ListBox на странице
                petNameList.DataSource = carPetNames;
                petNameList.DataBind();
        }
}
```

То, что должно получиться, представлено на рис. 14.48.

Рис. 14.48. Привязываем данные к элементам управления WebForm

Код приложения ASPData можно найти в подкаталоге Chapter 14.

Все массивы .NET происходят от единого общего предка — класса System.Array, а в классе System.Array реализован интерфейс IEnumerable. Мы говорим это к тому, что любой класс, в котором реализован этот интерфейс, может быть привязан к элементу управления WebForm (да и Windows Forms) в качестве источника данных. Например, если данные находились в объекте ArrayList, все будет точно гак же:

```
protected void Page_load(object sender, EventArgs e)
{
        if(!IsPostBack)
        {
                // Теперь вместо обычного массива у нас - объект ArrayList
                ArrayList carPetNames = new ArrayList();
                carPetNames.Add("Viper");
                carPetNames.Add("Ottis");
                carPetNames.Add("Alphonzo");
                carPetNames.Add("Cage");
                carPetNames.Add("TB");
                petNameList.DataSource = CarPetNames:
                petNameList.DataBind();
        }
}
```

Результат, естественно, остался тем же самым.

Элементы управления для проверки вводимых пользователем данных

Последняя разновидность элементов управления WebForm — это элементы управления, которые применяются для проверки вводимых пользователем данных. Наиболее важные элементы управления этого типа представлены в табл. 14.15,

Таблица 14.15. Элементы управления для проверки данных

Элемент управления	Описание
CompareValidator	Сравнивает значение, введенное в один элемент управления, со значением, введенным во второй элемент управления
CustomValidator	Позволяет определить пользовательский метод, при помощи которого и будет производиться проверка
RangeValidator	Определяет, попадает ли введенное пользователем значение в определенный диапазон
RegularExpressionValidator	Проверяет введенное значение на соответствие подстановочному выражению
RequiredFieldValidator	Позволяет убедиться, что в соответствующий элемент управления действительно введено значение (оно не оставлено пустым)
ValidationSummary	Отображает все ошибки, обнаруженные при проверке ввода, в виде списка, маркированного списка или обычного абзаца. Ошибки могут отображаться на web-странице или в специальном окне оповещения браузера

Давайте создадим новый проект C# на основе шаблона Web Application и назовем его ValidateWebApp. Изменим имя файла *.aspx на default.aspx, а затем откроем этот файл как графический шаблон времени разработки (чтобы можно было помещать элементы управления). Затем при помощи перетаскивания элементов управления из Toolbox создадим интерфейс, похожий на представленный на рис. 14.49. Все текстовые надписи можно сделать при помощи обычного кода HTML — использовать объект Label не обязательно.

А теперь посмотрим, как можно проверять правильность вводимых пользователем данных при помощи элементов управления WebForm. Например, нам нужно убедиться, что текстовое поле txtEMail пользователь не оставил пустым. Проверка

будет производиться при нажатии кнопки Submit, то есть при попытке пользователя отправить данные на сервер. Самый простой способ реализовать такую проверку — воспользоваться элементом управления RequiredFieldValidator.

Рис. 14.49. Простой интерфейс для ввода данных

Добавим RequiredFieldValidator на страницу при помощи Toolbox и откроем его свойства (рис. 14.50). Свойств, которые обязательно нужно настроить, два; ControlToValidate (здесь нужно выбрать имя текстового поля, обязательного для заполнения, — в нашем случае txtEMail) и ErrorMessage — сообщение об ошибке, которое будет выдаваться пользователю.

Вот и все. Если мы посмотрим, какой код в файле *.aspx сгенерировала для нас среда разработки Visual Studio.NET, то он будет таким:

```
<asp:RequiredFieldValidator id=RequiredFieldValidator1 style="Z-INDEX: 109:
LEFT: 351px; POSITION: absolute; TOP: 204px"
runat="server"
ErrorMessage="We need your e-mail Address!"
ControlToValidate="txtEMail">
</asp:RequiredFieldValidator>
```

Кроме того, в производном от Page классе C# для нашей страницы ASP.NET (в том файле, который указан в атрибуте Codebehind) появится новая переменная:

```
public class WebForm1 : System.Web.UI.Page
{
        protected System.Web.UI.WebControls.Button btnSubmit;
        protected System.Web.UI.WebControls.RequiredFieldValidator
                                                RequiredFieldValidator1;
        protected System.Web.UI.WebControls.TextBox txtEMail;
        protected System.Web.UI.WebControls.TextBox txtLName;
        protected System.Web.UI.WebControls.TextBox txtFName;

...
}
```

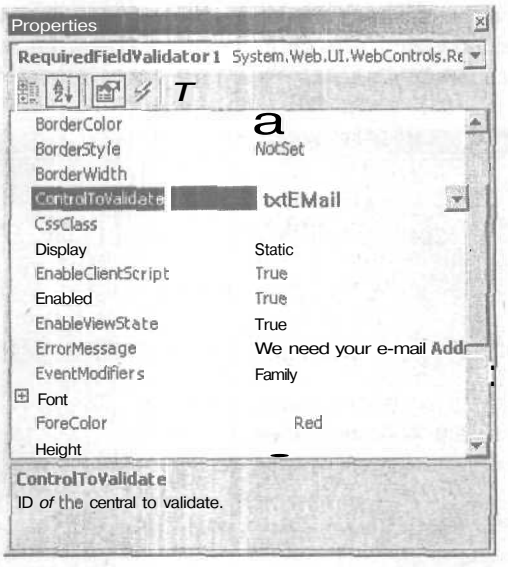


Рис. 14.50. Свойства объекта RequiredFieldValidator

Рис. 14.51. RequiredFieldvalidator в действии

Сохраним страницу и обновим ее в окне браузера. С виду ничто не изменится. Однако если мы попробуем нажать кнопку Submit, не заполнив поле txtEMail, то на странице рядом с этим полем появится сообщение об ошибке (рис. 14.51).

Если мы введем данные в текстовое поле и нажмем Submit, надпись исчезнет. Если в окне браузера просмотреть код web-страницы, то мы заметим в нем новую функцию JavaScript, которая была создана для нас без нашего участия. На самом деле все еще интереснее: если элемент управления WebForm обнаружит, что браузер не поддерживает работу со скриптами, то логика проверки вводимых данных будет реализована для выполнения на сервере!

Код приложения Validate Web App можно найти в подкаталоге Chapter 14.

Обработка событий элементов управления WebForm

События, генерируемые элементами управления WebForm, можно перехватывать и обрабатывать двумя способами. Первый способ — делать все непосредственно в браузере клиента при помощи клиентских браузерных скриптов JavaScript. Это традиционный подход, который наиболее удобен в тех ситуациях, когда нужно выполнять форматирование на web-странице, выводить оповещения в окне браузера или осуществлять прочие взаимодействия с объектной моделью, реализованной в браузерах. Но элементы управления WebForm предлагают нам и другой способ — обрабатывать и перехватывать их события *на сервере*. Для этого достаточно добавить обработчик события при помощи окна Property свойств элемента управления. Обычно такой способ наиболее удобен для выполнения операций, не связанных с графическим интерфейсом — например, для производства каких-то вычислений, редактирования таблицы с данными и т. п.

Давайте рассмотрим применение обработки событий элементов управления WebForm на сервере. В качестве элемента управления у нас будет использован Calendar, а реагировать мы будем на событие SelectionChanged. Выглядеть это может так:

```
protected void Calendar1_SelectionChanged (object sender. System.EventArgs e)
{
        Response.Write("<h5>Your car will be delivered on:" + Calendar1.Selecteddate.date
                                                                    +"</h5>");
}
```

Теперь при выборе пользователем даты в календаре сработает событие SelectionChanged и в окне браузера появится надпись, показанная на рис. 14.52.

Конечно, мы можем использовать для подобных целей и клиентские браузерные скрипты — никто этого нам не запрещает. Все зависит от ситуации — свои достоинства есть у каждого способа. Клиентским скриптам не нужно обращаться на сервер, поэтому они будут работать гораздо быстрее. С другой стороны, серверные скрипты проще и надежнее.

Подведение итогов

Создание web-приложений требует во многом иных подходов, чем создание обычных настольных приложений. В самом начале этой главы были очень бегло рассмотрены основные элементы, без которых не обойтись ни одному web-приложению: теги HTML, протокол HTTP, клиентские браузерные скрипты и серверные скрипты классических ASP.

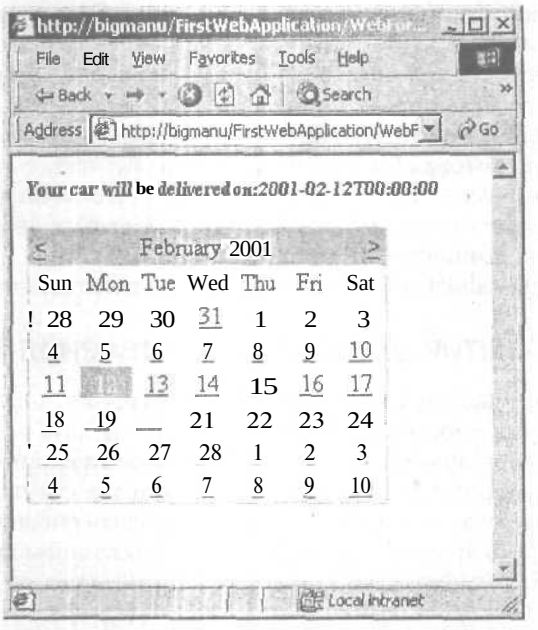


Рис. 14.52. Обработка событий элементов управления WebForm на сервере

Большая часть этой главы была посвящена созданию приложений ASP.NET. Каждому шаблону HTML нашего приложения — файлу *.aspx соответствует в ASP.NET класс, производный от System.Web.UI.Page. Работать с этими классами можно средствами привычных языков программирования .NET, например C#. Таким образом, в ASP.NET теперь можно использовать технологии объектно-ориентированного программирования, создавая код, пригодный для повторного использования. Мы последовательно рассмотрели основные свойства объекта Page (Session, Application, Request и Response), которые обеспечивают доступ к внутренним объектам класса, производного от Page. В самом конце главы были рассмотрены элементы управления WebForm — элементы управления, во многом аналогичные стандартным элементам управления Windows Forms, с помощью которых можно избежать трудоемкой и утомительной обязанности создавать теги HTML и клиентские скрипты вручную.

Web-службы 15

Во многих отношениях эта глава объединяет в единое целое многое из того, что мы узнали из предыдущих глав этой книги. Речь пойдет о web-службах ASP.NET – модулях кода .NET (обычно установленных на сервере IIS), к которому возможно удаленное обращение по протоколу HTTP.

Как мы увидим, web-службы строятся на основе трех взаимосвязанных технологий: Web Service Description Language (WSDL, язык описания web-служб), протокола подключения (HTTP-GET, HTTP-POST и SOAP) и службы обнаружения (discovery service). Как обычно, вначале мы поработаем с элементарным примером, создав web-службу, которая выполняет роль калькулятора, а затем создадим более изощренный пример web-службы, связанной с миром автомобилей, которая будет возвращать объекты ADO.NET DataSet, массивы ArrayList и пользовательские типы.

После того как мы научимся создавать web-службы, мы обратимся к созданию прокси-классов (при помощи Visual Studio.NET и утилиты wsdl). Эти прокси-классы позволят обращаться к web-службе как клиентам Web, так и другим клиентам, в том числе с помощью консольных приложений и обычных приложений Windows Forms.

Роль web-служб

Если посмотреть на web-службу «с высоты птичьего полета», то это — всего лишь блок кода, к которому можно обратиться по протоколу HTTP. Однако сама по себе эта формулировка значит уже очень многое. Подавляющее большинство используемых в настоящий момент технологий удаленной активации кода привязаны к конкретным протоколам (при этом требующих постоянных и надежных соединений), платформам и языкам программирования. В DCOM для обращения к удаленным типам COM используется требующий высокоскоростных надежных соединений RPC. В CORBA используется несколько протоколов, но все они также

требуют постоянного подключения и надежных соединений. EJB (Enterprise Java Beans) требует использования определенного протокола плюс языка программирования Java.

.NET сильно отличается от всех этих технологий. Прежде всего, как мы много раз могли убедиться, .NET обеспечивает большую степень языковой независимости, чем что-либо другое. Мы можем создавать при помощи C#, VB.NET или любого другого языка программирования для работы с .NET типы, к которым можно будет обращаться из клиента на любом .NET-совместимом языке. Кроме того, для обращения к web-службам ASP.NET нам нужно, чтобы на данной конкретной платформе был реализован протокол HTTP — и все! При всем существующем разнообразии платформ и операционных систем вряд ли найдется платформа, на которой не был бы реализован HTTP.

Таким образом, как web-разработчик, вы обнаружите, что для создания web-службы ASP.NET вы сможете использовать привычный и любимый язык программирования. Как клиента web-служб, думаю, вас обрадует тот факт, что для вызова методов типов web-служб вполне можно обойтись стандартным HTTP. Кроме того, как мы вскоре обнаружим, во взаимодействии с протоколом HTTP вы сможете также использовать стандартные XML и SOAP (Simple Object Access Protocol), что также немаловажно.

Web-служба строится из тех же типов, что и любая сборка .NET: из классов, интерфейсов, перечислений и структур, которые играют для клиента роль «черного ящика», отвечающего на запросы. Единственное важное ограничение, о котором необходимо постоянно помнить, связано с тем, что web-службы предназначены для обработки *удаленных* вызовов и поэтому в них следует избегать применения типов для работы с графическим интерфейсом. Web-службы предназначены для другого: они должны уметь выполнить какое-либо действие по запросу пользователя (произвести вычисления, считать данные из источника) и ждать следующего запроса.

Еще один важный момент, связанный с web-службами, который обязательно необходимо осознать, состоит в том, что для них вовсе не обязательно использовать клиентов, работающих через браузер. К web-службе вполне могут обращаться и обычные консольные или Windows-клиенты (локальные, клиенты терминальных служб и т. п.). Для этого в .NET предусмотрены специальные средства, которые позволяют генерировать так называемые прокси-сборки. Мы обращаемся к типам этой прокси-сборки, как к обычному типу .NET, а она уже перенаправляет запрос в web-службу (при помощи HTTP или сообщений SOAP) и возвращает клиенту полученные результаты.

Инфраструктура web-службы

Web-службе (как и обычному приложению ASP.NET) обычно соответствует виртуальный каталог на сервере IIS. Однако для web-службы вам потребуется также реализовать дополнительную поддерживающую инфраструктуру. К ней относятся:

- протокол подключения (HTTP-GET, HTTP-POST или SOAP);
- служба описания — description service (чтобы клиент мог получить информацию о том, что делает эта web-служба);

- служба обнаружения — discovery service (чтобы клиент мог получить информацию о том, что web-служба существует).

В этой главе нам предстоит реализовать каждую часть инфраструктуры в ваших примерах. А пока — краткое описание.

Протокол подключения

Б web-службах ASP.NET (как и в ADO.NET) стандартный формат передачи информации между службой и клиентом — это формат XML. Сама передача происходит при помощи протокола HTTP. Мы можем использовать различные методы передачи информации — метод HTTP GET, HTTP POST и SOAP. Ориентироваться следует на SOAP, поскольку при помощи этого протокола мы можем обеспечить передачу очень сложных типов (пользовательских классов, объектов ADo.NET DataSet, массивов объектов и т. п.).

Служба описания

При обращении к удаленной web-службе клиент должен обладать полной информацией о членах типов web-службы, которые предоставлены в его распоряжение. Например, клиент должен иметь информацию о том, что он может вызвать метод Foo(), а также все необходимые параметры этого метода: какие параметры этот метод принимает и что он возвращает. За предоставление клиенту этой информац] ии и ответственна служба описания (description service) web-службы. Как обычно, сама информация о web-службе предоставляется в формате XML XMLschema, которая используется для описания web-службы, называется WSDL (Web Service Description Language, WSDL).

Служба обнаружения

Служба обнаружения (discovery service) позволяют клиенту обнаруживать web-службы по адресу URL. Для этой службы используются файлы *.disco (от discovery), опять-таки в формате XML. Мы познакомимся с синтаксисом этих файлов в последней части этой главы.

Обзор пространств имен web-служб

Как вы уже, наверное, догадываетесь, разработчики .NET заготовили для нас множество типов, которые могут быть использованы как для построения самих web-служб, так и для создания необходимой инфраструктуры. Эти типы определены в пространствах имен, представленных в табл. **15.1,**

Таблица 15.1. Пространства имен для web-служб

Пространство имен	Описание
System.Web.Services	В этом пространстве имен определен минимально достаточный набор типов для построения web-службы
System.Web.Services.Description	Набор типов для программного взаимодействия с WDSL

продолжение

Таблица 15.1 (*продолжение*)

Пространство имен	Описание
System.Web.Services.Discovery	Эти типы обеспечивают клиенту web-служб возможность программно обнаруживать web-службы, установленные на конкретном компьютере. Используются вместе с файлами *.disco
System.Web.Services.Protocols	Между web-службой и клиентом данные могут передаваться по протоколам HTTP GET, HTTP POST и SOAP. В этом пространстве имен определены типы, которые предназначены для работы с этими протоколами

Пространство имен System.Web.Services

В большинстве проектов по созданию web-служб единственные типы, с которыми нам придется взаимодействовать напрямую — это типы пространства имен System.Web.Services. Этот набор типов не так уж велик: типы System.Web.Services представлены в табл. 15.2.

Таблица 15.2. Типы пространства имен System.Web.Services

Тип	Описание
WebMethodAttribute	Добавление атрибута [WebMethod] в определение метода web-службы означает, что этот метод может быть вызван удаленным клиентом по HTTP
WebService	Определяет необязательный базовый класс для web-службы
WebServiceAttribute	Этот атрибут может быть использован для размещения информации о web-службе (например, для строки, описывающей ее возможности). Атрибут является необязательным
WebServiceBindingAttribute	Объявляет связывающий протокол, который реализован методом web-службы

Пример элементарной web-службы

Перед тем как приступать к подробностям, давайте сформируем мысленный образ того, о чем идет речь, и создадим очень простую web-службу. Для этого запустим Visual Studio.NET и создадим новый проект C# на основе шаблона Web Service (рис. 15.1). Мы назовем этот проект CalcWebService.

Как для любого проекта ASP.NET, при создании проекта Web Service Visual Studio.NET автоматически создаст виртуальный каталог для этого проекта на сервере IIS (рис. 15.2), а ненужные в этом каталоге файлы проекта *.sln и *.suo разместит в подкаталоге \MyDocuments\Visual Studio Projects.

Если вы собираетесь при изучении этой главы использовать уже готовый исходный код, проще всего создать новый проект и импортировать туда готовые классы (чтобы не возиться с созданием виртуальных каталогов на IIS).

Что же создала для нас Visual Studio? Набор файлов, сгенерированных автоматически, представлен в окне Solution Explorer на рис. 15.3.

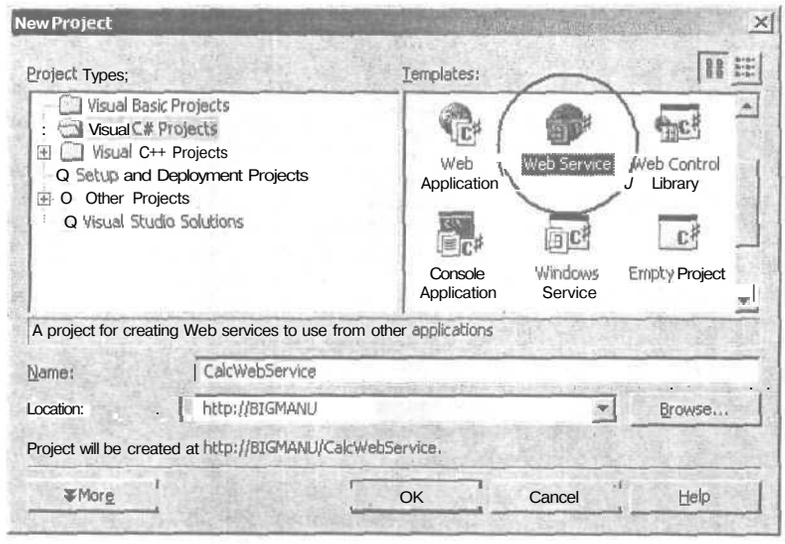


Рис. 15.1. Создаем новый проект Web Service

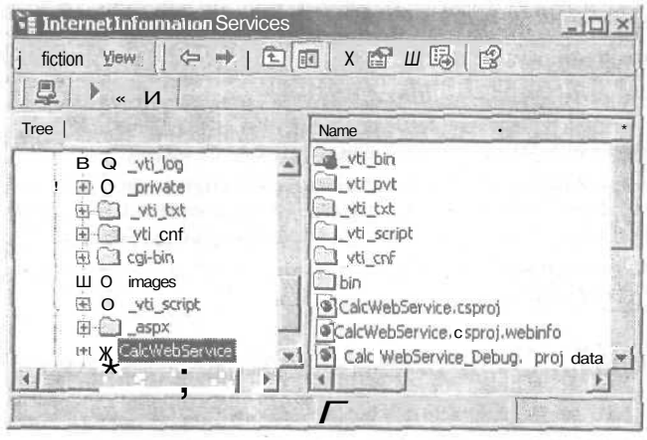


Рис. 15.2. Для web-службы будет автоматически создан виртуальный каталог на сервере IIS

С подавляющим большинством этих файлов мы уже познакомились в главе 14 — они общие для всех приложений ASP.NET. Вкратце перечислим назначение наиболее важных из них.

Файл Global.asax предназначен для организации реагирования на события глобального уровня (общие для всех сеансов подключения). Файл Web.config позволяет нам в формате XML определить основные параметры приложения ASP.NET (в данном случае нашей web-службы). Вся реальная работа у нас (как и в большинстве реальных проектов) будет производиться исключительно с тремя файлами: *.asmx, *.asmx.cs и *.disco.

Рис. 15.3. Исходный набор файлов проекта

Таблица 15.3. Наиболее важные файлы проекта Web Service

Файл	Описание
*.asmx	Файл в XML-совместимом формате WSDL, в котором содержится информация о методах, предоставляемых пользователю, и на основе которого среда выполнения ASP.NET автоматически генерирует код HTML
*.amsx.cs	Обычный исходный файл C#, в котором хранятся определения методов, предоставляемых пользователям web-службы. Файл *.asmx ссылается на соответствующий ему файл *.asmx,cs при помощи атрибута Codebehind
*.disco	XML-совместимый файл с описанием web-служб, которые можно найти по указанному адресу URL

Исходный файл C# для web-службы (*.asmx.cs)

Как уже говорилось Р предыдущих главах, одно из наиболее важных преимуществ ASP.NET — это возможность создавать web-приложения при помощи полнофункциональных объектно-ориентированных языков программирования (в нашем случае C#), а не ограничиваться скриптами (как в классических ASP). Файл *.asmx можно представить себе как шаблон, на основе которого среда выполнения ASP.NET генерирует код HTML, передаваемый в браузер клиента. Для файла *.asmx при помощи атрибута Codebehind («код за сценой») можно определить файл на «нормальном» языке программирования, в котором и будет реализована вся программная логика web-службы. В нашем случае для web-службы был выбран проект C#, поэтому и файл Codebehind — *.asmx.cs, исходный файл C#.

Что же вложила в этот файл C# среда выполнения Visual Studio.NET? А вот что:

```
public class Service1 : System.Web.Services.WebService
{
```

```
        public Service() { InitializeComponent(); }
        private void InitializeComponent() {}
        public override void Dispose() {}
};
```

Ничего интересного, за одним существенным исключением: для класса C# в качестве базового был выбран класс System.Web.Services.WebService. Сразу оговоримся, что это совершенно необязательно. Мы вполне можем создать web-службу, обойдясь и без этого базового класса. Определение будет тем же самым, только наш класс будет теперь производиться напрямую от System.Object, а замещенный метод Dispose() лучше закомментировать:

```
// А я все равно web-служба
public class Service1
{
        public Service0 { InitializeComponent(); }
        private void InitializeComponent() {}
        // public override void Dispose0 {}
};
```

Как мы сможем убедиться, функциональности нашей web-службы это нисколько не повредит. Однако выбор в качестве базового класса WebService автоматически обеспечивает нашей web-службе очень полезный набор членов, с которым мы вскоре познакомимся.

Реализуем методы web-службы

В нашей первой web-службе мы ничего не будем усложнять и ограничимся четырьмя методами, при помощи которых пользователь сможет производить элементарные арифметические операции. Все эти методы будут доступны по HTTP, но чтобы среда выполнения ASP.NET поняла, какие методы нужно выкладывать по HTTP для пользователей, эти методы нужно пометить атрибутом [WebMethod]. В общем, определение нашего класса Service мы сделаем таким:

```
public class Service1 : System.Web.Services.WebService
{
        public Service0 { InitializeComponent(); }
        private void InitializeComponent() {}
        public override void Dispose0 {}

        [WebMethod]
        public int Add(int x, int y) { return x + y; }

        [WebMethod]
        public int Subtract(int x, int y) { return x - y; }

        [WebMethod]
        public int Multiply(int x, int y) { return x * y; }

        [WebMethod]
        public int Dividednt x, int y)
        {
                if(y = - 0)
                {
                        throw new DivideByZeroException("Dude, can't divide by zero!");
                }
```

```
            return x / y;
        }
    }
```

Работа клиента с web-службой

После того как web-служба будет откомпилирована, запустим ее на выполнение (можно прямо в Visual Studio). По умолчанию в качестве клиента будет открыто окно нашего браузера, а в нем откроется страница HTML со списком всех методов, которые мы пометили атрибутом [WebMethod] — см. рис. 15.4.

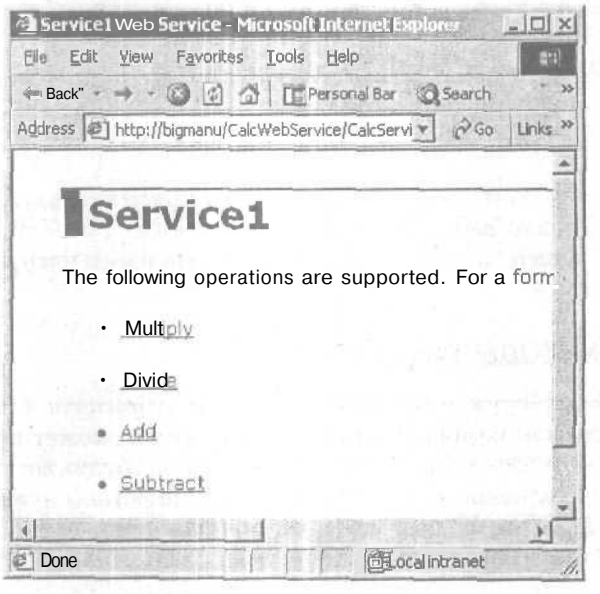


Рис. 15.4. Клиент подключился к нашей web-службе

Конечно же, мы можем не только просматривать список методов web-службы, но и вызывать его прямо из браузера — заботливая среда выполнения ASP.NET позаботилась и об этом! Например, перейдем по гиперссылке на Add и введем в текстовые поля значения (рис. 15.5). Осталось только нажать кнопку Invoke, и среда выполнения ASP.NET вызовет метод, передаст ему введенные значения и вернет нам результат в формате XML (рис. 15.6).

Конечно же, метод можно вызывать и не используя графический интерфейс. Например, если мы подсмотрим, какой запрос был отправлен в web-службу, то он будет выглядеть так:

```
http://Имя_хоста/CalcWebService/CalcService.asmx/Add?x=44&y=446
```

Как мы видим, запрос состоит из адреса страницы *.asmx с добавлением имени метода и парами имя — значение, представляющими параметры метода.

Как мы только что убедились, создать web-службу в ASP.NET — это очень просто. Нам еще предстоит поработать с примером более серьезной службы, однако пока мы обратимся к некоторым особенностям архитектуры web-служб ASP.NET.

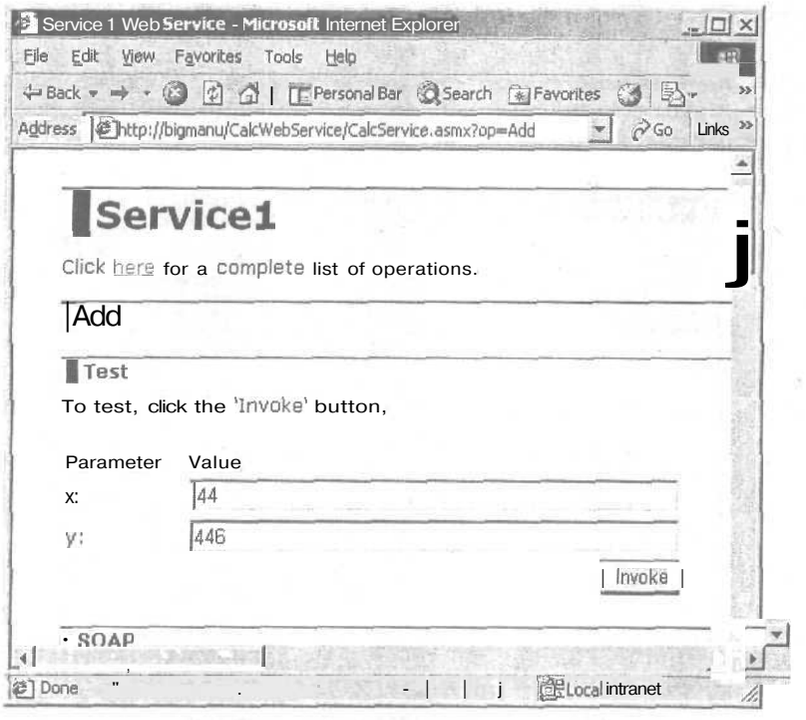


Рис. 15.5. Среда выполнения ASP.NET автоматически сгенерирует текстовые поля для ввода параметров метода и прочий необходимый код

Рис. 15.6. Результат вычислений, возвращаемый в формате XML

Тип WebMethodAttribute

Атрибут WebMethod (представленный типом WebMethodAttribute) должен обязательно быть указан для каждого метода web-службы, предоставляемого в распоряжение клиента. Как большинство других атрибутов .NET, для атрибута WebMethod можно использовать дополнительные параметры (которые на самом деле являются параметрами конструктора WebMethodAttribute). Например, мы можем предоставить клиенту дополнительную информацию о методе web-службы:

```
[WebMethod(Description = "Yet another way to add members!")]
public int Add(int x. int y){ return x + y; }
```

Как мы видим, параметр Description атрибута WebMethod очень похож на атрибут [helpstring] в IDL. А результатом его применения будет дополнительная информация о методе на web-странице (рис. 15.7).

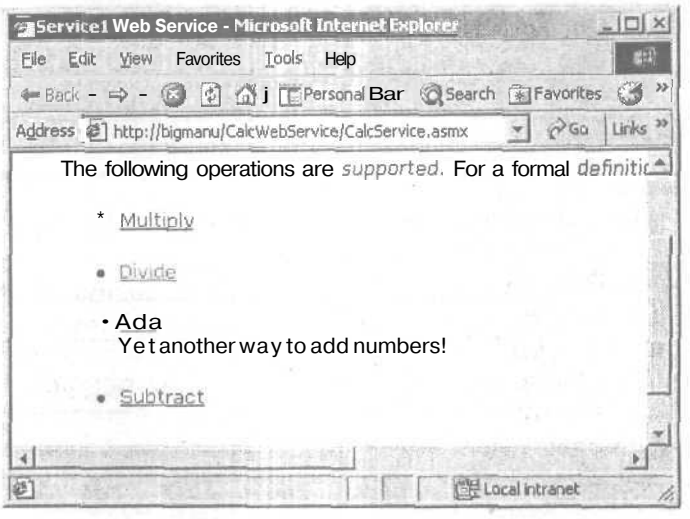


Рис. 15.7. Параметр Description в действии

Если поинтересоваться, что же происходит при добавлении к методу параметра Description, то окажется, что в коде WSDL (об этом — позже) файла *.asmx появился дополнительный тег <documentation>:

```
<operataion name="Add">
        <input message="so:AddSoapIn" />
        <output message="so:AddSoapOut" />
        <documentation>Yet another way to add numbers!</documentation>
</operation>
```

Параметр Description — не единственный, который можно использовать для атрибута WebMethod. Наиболее важные параметры этого атрибута представлены в табл. 15.4.

Таблица 15.4. Параметры атрибута WebMethod

Параметр	Описание
Description	Позволяет добавить дружественное описание для метода web-службы
EnableSession	При установленном значении true (по умолчанию) для метода можно использовать данные сеанса подключения (например, определять, сколько раз в течение сеанса он был вызван пользователем)
MessageName	Этот параметр можно использовать, чтобы определить, как метод web-службы будет представлен в коде WSDL. Обычно он применяется для того, чтобы не допустить конфликты имен
TransactionOption	Методы web-службы могут использоваться в качестве корня транзакции COM+. Для этого параметра используются значения из перечисления System. EnterpriseServices.TransactionOption

Давайте рассмотрим на примере, для чего нужен параметр MessageName. Предположим, что в нашем web-калькуляторе появился дополнительный метод Add() для сложения двух значений с плавающей запятой:

```
[WebMethod(Description = "Add 2 integers.")]
public int Add(int x, int y) { return x + y; }

[WebMethod(Description = "Add 2 floats.")]
puDlic float Add(float x, float y) { return x + y; }
```

Если мы попробуем откомпилировать наш проект, то никаких ошибок не будет — с точки зрения C# все в порядке. Однако если мы попытаемся обратиться к web-службе из браузера, мы увидим не совсем то, на что рассчитывали:

```
System.Exception: Both Single Add(Single, Single) and Int32 Add(int32, Int32) use the
message name 'Add'.
(System.Exception: И метод Single Add(Single, Single), и метод Int32 Add(int32, Int32)
используют имя сообщения 'Add'.)
```

Конфликт имен произошел на уровне WSDL: необходимо, чтобы для каждого атрибута <soap:operation soapAction> (который используется для идентификации метода web-службы) использовалось уникальное значение (то есть имя web-метода). Однако по умолчанию значением этого атрибута считается имя метода в определении этого метода в C#. Чтобы решить проблемы с конфликтом имен, достаточно для одного из методов добавить новое значение параметра MessageName:

```
[WebMethod(Description = "Add 2 integers.")]
public int Add(int x, int y) { return x + y; }

[WebMethodCDescrlptIon = "Add 2 floats.", MessageName = "AddFloats")]
public float Add(float x, float y) { return x + y; }
```

После этого идентификаторы методов в WSDL станут разными:

```
<operation name="Add">
<soap:operation soapAction="http://tempuri.org/AddFloats" style="document" />
        <input name="AddFloats">
                <soap:body use="literal" />
        </input>
        <output name="AddFloats">
                <soap:body use="literal" />
        </output>
</operation>

<operation name="Add">
<soap:operation soapAction="http://tempuri.org/Add" style="document" />
        <input>
                <soap:body use="literal" />
        </input>
        <output>
                <soap:body use="literal" />
        </output>
</operation>
```

Если нам потребуется использовать описание для всей нашей web-службы, а и е для отдельного метода, можно использовать для класса C# атрибут WebService с аналогичным параметром Description, например, так:

```
[WebService(Description = "The painfully simple web service" )]
public class Service1 : System.Web.Services.WebService
{
...
}
```

Если мы запустите нашу web-службу после этого добавления, результат будет таким, как показано на рис. 15.8.

Рис. 15.8. Параметр Description атрибута WebService определяет описание для web-службы в целом

Базовый класс System.Web.Services.WebService

Как мы уже заметили, по умолчанию Visual Studio.NET производит класс нашей web-службы от базового класса System.Web.Services.WebService. В принципе web-служба будет работать и без этого класса в качестве базового, однако в WebService предусмотрен набор очень полезных членов и вложенных типов, которые обеспечивают web-службе те же возможности, которые имеются у стандартного приложения ASP.NET. Обычно в программном коде приходится взаимодействовать со свойствами класса web-службы, унаследованными от WebService. Наиболее важные свойства WebService представлены в табл. 15.5.

Таблица 15.5. Наиболее важные свойства класса WebService

Свойство	Описание
Application	Возвращает ссылку на объект HttpApplicationState для текущего запроса HTTP
Context	Возвращает ссылку на объект HttpContext. Этот объект можно использовать для получения разнообразной информации о контексте выполнения запроса на сервере IIS
Server	Возвращает ссылку на объект HttpServerUtility, который можно использовать для получения информации о сервере, на котором выполняется запрос
Session	Возвращает ссылку на объект HttpSessionState. Используется для получения информации о текущем сеансе подключения
User	Возвращает объект ASP.NET User, который может быть использован для получения информации о пользователе, работающем с web-службой, или, например, для целей аутентификации

Web Service Description Language (WSDL)

Во всех программных технологиях, использующих межъязыковое взаимодействие, используются специальные средства для описания программных модулей и типов в них. В COM для этого применяется язык IDL, в обычных приложениях .NET - метаданные сборки (манифест) и типов. Для web-служб ASP.NET такое специальное средство также предусмотрено: это — WSDL.

WSDL (Web Service Description Language, язык описания web-служб) — это XML-совместимый язык, который полностью описывает для внешних клиентов возможности web-служб, методы, которые клиенты могут вызвать, а также поддержку протоколов подключения к web-службам (HTTP-GET, HTTP-POST и SOAP). Сразу скажем, что код WSDL в ASP.NET генерируется автоматически. Например, в нашем примере WSDL-описание web-службы можно получить прямо из окна браузера. Если мы откроем в браузере страницу CalcService.asmx, то в верхней части страницы будет гиперссылка Service Description (рис. 1.5.9). Перейдя по ней, можно прочитать код WSDL для нашей web-службы (рис, 15.10).

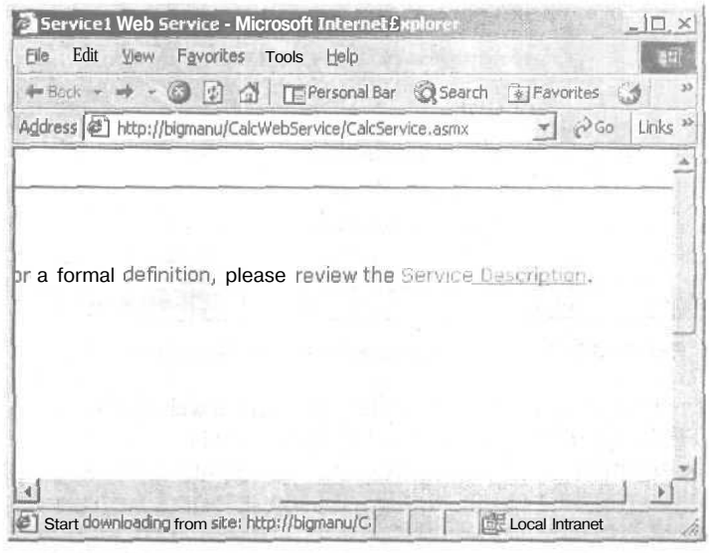


Рис. 15.9. Ссылка на описание web-службы в формате WSDL

Поскольку код WSDL генерируется автоматически, мы имеем полное право и не думать о том, что там написано в коде WSDL для нашей web-службы. Досконально разбирать все возможности WSDL мы не будем, но все же обратим внимание на наиболее принципиальные моменты.

Прежде всего, любое определение web-службы на WSDL начинается с тега <definitions>. Далее следуют ссылки на узлы, определяющие протоколы подключения к web-службе:

```
<?xml version="1.0" ?>
<definitions
xmlns:s="http://www.w3.org/2000/10/XMLSchema"
```

```
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:urt="http://microsoft.com/urt/wsdl/text/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s0="http://tempuri.org/" targetNamespace="http://tempuri.org/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
...
```

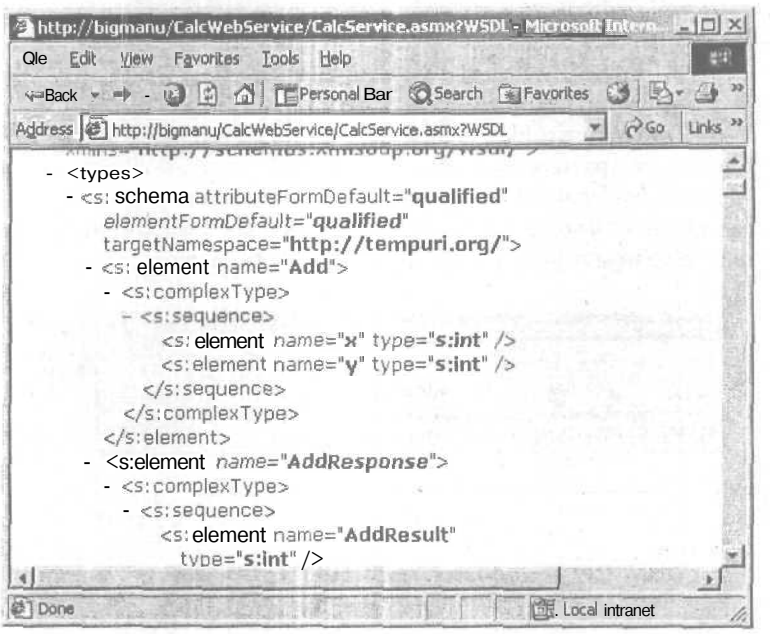


Рис. 15.10. А это — само описание web-службы в формате WSDL

За ними следуют определения WSDL для каждого web-метода в терминах протоколов HTTP-GET, HTTP-POST и SOAP (рис. 15.11).

Для каждого метода предусмотрены отдельные определения In (для приема данных) и Out (для возврата данных клиенту), при этом каждая пара перечисляется отдельно для каждого протокола подключения. Например, вот пара In/Out в WSDL-определении метода Subtract для протокола подключения HTTP-POST:

```
<message name="SubtractHttpPostIn">
      <part name="x" type"s:string" />
      <part name="y" type"s:string" />
</message>

<message name="SubtractHttpPostOut">
      <part name="Body" element="s0:int" />
</message>
```

А вот так выглядит та же пара того же метода Subtract для протокола SOAP:

```
<message name="SubtractSoapIn">
      <part name="parameters" element="s0:Subtract" />
</message>
```

```
<message name="SubtractSoapOut">
      <part name="parameters" element="s0:SubtractResponse" />
</message>
```

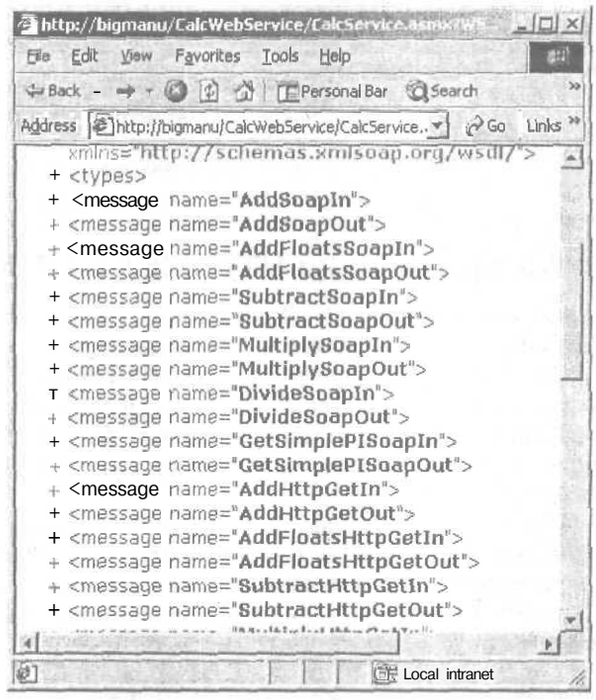


Рис. 15.11. Для каждого метода web-службы предусмотрены WSDL-определения для всех протоколов подключения

В WSDL, как и во многие другие метаязыки, глубоко вникать есть смысл в том случае, если вы собираетесь создать что-нибудь вроде своего собственного анализатора кода WSDL или просмотрщика типов ASP.NET. Если создание таких программных продуктов действительно входит в ваши планы, мы советуем вам также обратиться к типам пространства имен System.Web.Services.Description. Эти типы библиотеки базовых классов .NET позволяют удобно манипулировать кодом WSDL непосредственно из программы.

Протоколы подключения к web-службам

Как уже было сказано, основная задача любой web-службы ASP.NET — вернуть клиенту запрашиваемые им данные по протоколу HTTP. Однако для обмена данными между клиентом и сервером можно использовать три разных метода (они и называются протоколами подключения — wire protocols): HTTP-GET, HTTP-POST и SOAP. Мы уже неоднократно упоминали об этих протоколах, однако для удобства сведения о них мы на этот раз сведем в таблицу:

Выбор протокола подключения определяет то, какими типами (передаваемыми методам web-служб и возвращаемыми ими клиентам) смогут обмениваться кли-

ент и web-служба. Можно сделать лишь общее замечание о том, что наибольшие возможности обеспечивает протокол SOAP. Однако вначале мы рассмотрим применение протоколов HTTP-GET и HTTP-POST.

Таблица 15.6. Протоколы подключения к web-службам

Протокол подключения	Характеристика
HTTP-GET	При использовании этого метода данные добавляются к адресной строке URL
HTTP-POST	Данные добавляются в специальное поле заголовка HTTP
SOAP	Данные передаются в XML-совместимом формате SOAP

Обмен данными при помощи HTTP-GET и HTTP-POST

При передаче данных методом HTTP-GET данные дописываются к строке запроса в формате URL в виде пар имя — значение (сам адрес отделяется от набора значений вопросительным знаком (?)). При применении HTTP-POST данные передаются при помощи тех же пар имя — значение, только они помещаются в специальное поле заголовка протокола HTTP. При использовании обычных методов возвращаемый клиенту результат всегда возвращается в простом формате XML в виде `<имя_типа>Значение</имя_типа>`.

HTTP-GET и HTTP-POST — методы очень простые и многим разработчикам хорошо знакомые. Однако их возможности оставляют желать лучшего: с их помощью мы не можем передавать сложные данные, такие как структуры или экземпляры объектов. Фактически клиент и web-служба могут обмениваться только типами, представленными в табл. 15.7.

Таблица 15.7. Типы данных, которые можно передавать при помощи протоколов HTTP-GET и HTTP-POST

Тип	Комментарий
Перечисления	Передача объектов типов, производных от System.Enum, вполне возможна, но следует учитывать, что эти объекты передаются как обычные строковые значения
Простые массивы	Можно передавать только массивы примитивов (но не объектов пользовательских типов)
Строковые значения	Строковые значения передаются без проблем. При помощи строковых значений также передаются и значения других типов данных CLR, таких как Int16, Int32, Int64, Boolean, Single< Double, Decimal, DateTime и многих других

Передача данных при помощи HTTP-GET и HTTP-POST производится очень просто. Мы создаем web-страницу с формой HTML, а в качестве получателя для этой формы указываем файл *.asmx. Метод передачи данных определяется при помощи атрибута method тега <form>. Вот пример такой web-страницы (она будет называться HTMLPage1.htm), которая будет принимать от пользователя два значения и передавать их методу Subtract() нашей web-службы CalcWebService:

```
<HTML>
<HEAD>
<TITLE><?TITLE>
<META NAME="GENERATOR" Content=Microsoft Visual Studio 7.0">
```

```
</HEAD>
<BODY>

<form method = 'GET' action = 'http://localhost/CalcWebService/CalcService.asmx/Subtract'>
       <p>First Number:
       <input id=Text1 name = x type=text> </p>
       <p>Second Number
       <input id=Text2 name = y type=text> </p>
       <p>
       <input id=Submit1 type=submit value=Submit> </p>
</form>

</BODY>
</HTML>
```

То, как выглядит эта страница (можно сказать, пользовательский интерфейс нашей web-службы), показано на рис. 15.12.

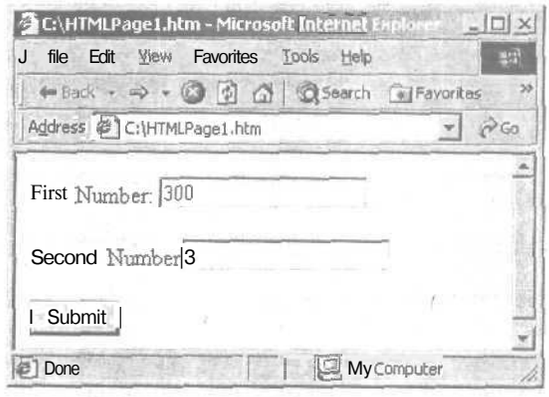


Рис. 15.12. Форма HTML для передачи данных в web-службу

Код такой простой, что комментировать почти нечего. Отметим только два момента. Обратите внимание, что мы в качестве получателя данных формы указали не только адрес страницы *.asmx, но и имя вызываемого метода — это вполне допускается. Кроме того, для каждого из текстовых полей мы при помощи атрибута name определили имя параметра, которому оно соответствует (x и y).

Результат произведенных web-службой вычислений представлен на рис. 15.13.

Рис. 15.13. Результат вычислений web-службы

Обратите внимание на адресную строку браузера: ее формат — верное доказательство, что при передаче клиенту данных использовался метод HTTP-GET.

Web-страницу CalcGET можно найти в подкаталоге Chapter 15.

Обмен данными при помощи SOAP

Гораздо более привлекательная альтернатива методам HTTP-GET и HTTP-POST - использование протокола подключения SOAP. Отличительной особенностью этого протокола является то, что с его помощью мы можем обмениваться сложными типами данных. Мы можем передавать все те же типы данных, что и с помощью HTTP-GET и HTTP-POST, плюс дополнительные, которые представлены в табл. 15.8, Вся передача информации производится в XML-совместимом формате.

Таблица 15.8. Типы, которые можно передавать при помощи SOAP

Тип	Комментарий
ADO..NET DataSet	В принципе DataSet можно отнести к «пользовательским типам», но этот тип так важен для обмена данными в реальных приложениях, что мы вынесли его в отдельную строку
Сложные массивы	Можно передавать массивы классов, структур и узлов XML
Пользовательские типы	С помощью SOAP можно переносить объекты пользовательских классов, структуры и узлы XML
Узлы XML	С помощью SOAP можно передавать любые узлы XML!

Подробное рассмотрение SOAP не входит в наши планы, однако наиболее важные моменты мы все же отметим.

В первую очередь необходимо сказать о том, что SOAP изначально создавался как очень простой в обращении протокол. Кроме того, он, как и все, что связано с XML, абсолютно независим от платформ, операционных систем, языков программирования и протоколов передачи данных. Например, данные SOAP мы можем передавать с помощью практически любых Интернет-протоколов (HTTP, SMTP и т. п.).

Любое описание в формате SOAP имеет два аспекта: информация, относящаяся к самому сообщению в целом, и данные в формате XML, относящиеся к составным частям данного сообщения,

Например, при использовании протокола SOAP для вызова метода Add() в нашем примере определение этого метода в SOAP будет выглядеть так:

```
<message name="AddFloatsSoapIn">
      <part name="parameters" element="s0:AddFloats" />
</message>
<message name="AddFloatsSoapOut">
      <part name="parameters" element="s0:AddFloatsResponse" />
</message>
```

Откроем код WSDL для нашей web-службы Calc Web Service. Ближе к концу страницы мы можем найти три узла XML, которые описывают привязки (bindings) нашей web-службы к протоколам HTTP-GET, HTTP-POST и SOAP (рис. 15.14). Эти записи определяют, что наша web-служба будет работать по каждому из этих трех протоколов.

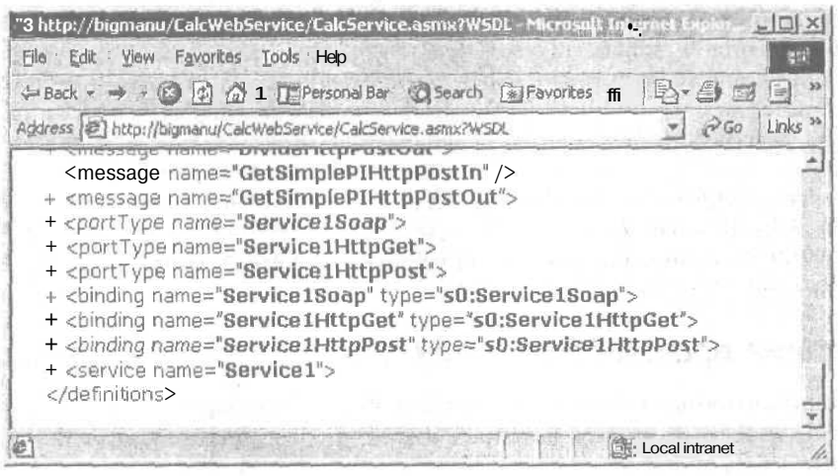


Рис. 15.14. Привязки web-службы к протоколам подключения

Если открыть узел для привязки SOAP, там можно будет найти следующий код для метода Ada() (обратите внимание на теги <1nput> и <output>).

```
<operation name="Add">
<soap:operation soapAction="http://tempuri.org/AddFloats" style="document" />
        <input name="AddFloats">
                <soap:body use="literal" />
        </input>
        <output name="AddFloats">
                <soap:body use="literal" />
        </output>
</operation>
```

Было бы интересно разобраться с каждым тегом и атрибутом SOAP, однако, к сожалению, мы ограничены рамками данной книги. Возможно, вас утешит то, что спецификации SOAP найти совсем несложно (адреса URL с этой информацией вообще помещаются в каждый файл WSDL, как мы могли убедиться). Кроме того, на практике, как мы уже видели и увидим в следующих примерах, требуемый код SOAP генерируется Visual Studio.NET автоматически, и нет необходимости править его вручную.

Файл CalcService.asmx.cs можно найти в подкаталоге Chapter 15.

Прокси-сборки для web-служб

Как мы могли убедиться, и передача данных клиентом (при работе по SOAP), и получение их (при работе по любому протоколу подключения) производится в XML-совместимых форматах. Конечно, мы можем создавать клиента для web-служб с модулями компоновки кода в XML и модулями анализа возвращаемых узлов XML, но это довольно трудоемкое занятие. Кроме того, немного странно использовать на клиенте код, к примеру, C#, потом преобразовывать промежуточные результаты в XML, чтобы с их помощью вызывать опять-таки методы C#, но уже на web-службе. Как было бы удобно работать напрямую!

В общем, мечты уже стали реальностью. Рекомендуется не обращаться на web-службу из клиента напрямую (хотя этого никто тоже не запрещает), а вначале создать промежуточную прокси-сборку на привычном C# или другом языке программирования и обращаться именно к ней — а она уже будет сама перенаправлять запросы на web-службу, получать возвращаемые результаты и передавать их вызывающему клиенту (который может быть обычным клиентом Windows Forms, или консольным, или клиентом ASP.NET — каким угодно). Самое замечательное во всем этом то, что прокси-сборка создается автоматически — при помощи Visual Studio.NET или специализированной утилиты wsdl.exe. А теперь посмотрим, как это делается.

Создание прокси-сборки при помощи утилиты wsdl.exe

Один из возможных способов сгенерировать прокси-сборку — воспользоваться утилитой wsdl.exe, входящей в состав .NET SDK. Минимально необходимый синтаксис ее очень прост:

```
wsdl.exe /out:C\calcproxy.cs http://localhost/calcwebservice/calcservice.asmx?WSDL
```

Параметр /out — это, конечно, имя создаваемого исходного файла для прокси-сборки. Далее указывается адрес URL, по которому может быть получено описание web-службы в формате WSDL (см. например, гиперссылку на рис. 15.9). По умолчанию генерируется именно код C#, однако по нашему желанию может быть сгенерирован код и на Visual Basic.NET, и на JavaScript.NET. Дополнительные параметры командной строки для wsdl.exe приведены в табл. 15.9.

Таблица 15.9. Параметры командной строки утилиты wsdl.exe

Параметр	Описание
/l[anguage]:	Определяет язык, на котором будет сгенерирован исходный код для прокси-сборки. Допустимые параметры — CS (по умолчанию), VB и JS
/n[amespace]:	Определяет пространство имен для генерируемой сборки. По умолчанию будет использовано глобальное пространство имен
//out:	Определяет имя файла, в котором будет сохранен сгенерированный код. По умолчанию используется имя файла, создаваемое на основе имени web-службы
/protocol:	Определяет, по какому протоколу создаваемая прокси-сборка будет обращаться на web-службу. По умолчанию используется значение SOAP, можно также использовать значения HttpGet и HttpPost. Можно использовать также свой собственный пользовательский протокол, но тогда его надо будет определить в специальном файле конфигурации

Как выглядит исходный код генерируемой прокси-сборки

В отличие от множества других прокси-сборок, предусмотренных в .NET (для работы с COM, COM+ и т. п.), прокси-сборки для web-служб создаются в виде файлов с исходным кодом, а не откомпилированным. Поэтому будет особенно интересно заглянуть в генерируемый файл и посмотреть, что нами было создано. Выглядит все это так (обратите внимание на атрибут WebServiceBinding):

```
using System.Xml.Serialization;
using System;
```

```
using System.Web.Serices.Protocols;
using System.Web.Services;

[System.Web.Services.WebServiceBindingAttribute(Name="Service1Soap", Namespace="http://
                                                                    tempuri.org/")]
public class Service1 : System.Web.Services.Protocols.SoapHttpClientProtocol
{
        public Service1()
        {
                this.Url = "http://localhost/calcwebservice/calcservice.asmx";
        }
...
}
```

Как мы видим, информация о том, где находится web-служба, помещается при помощи свойства Url в объект класса прямо во время работы конструктора. Еще один важный момент: класс нашей прокси-сборки — это класс, производный от базового класса SoapHttpClientProtocol. В этом базовом классе и определены те возможности, с помощью которых наша прокси-сборка будет взаимодействовать с web-службой. Некоторые наиболее важные члены, унаследованные от SoapHttpClientProtocol, представлены в табл. 15.10.

Таблица 15.10. Наиболее важные унаследованные члены

Член	Описание
BeginInvoke()	Производит асинхронный вызов метода web-службы
EndInvoke()	Завершает асинхронный вызов метода web-службы
Invoke()	Производит синхронный вызов метода web-службы
Proxy	Позволяет получить или установить информацию о настройках прокси-сервера для обращения к web-службе через брандмауэр
Timeout	Позволяет получить или установить тайм-аут (в миллисекундах), в течение которого прокси-сборка будет ожидать возврата результатов при синхронном вызове метода web-службы
Url	Позволяет получить или установить адрес URL для web-службы
UserAgent	Позволяет получить или установить значение заголовка user agent для производимых запросов (то есть каким браузером будет представляться данная прокси-сборка web-службе)

В сгенерированной прокси-сборке автоматически генерируются определения методов для синхронного и асинхронного вызова каждого метода web-службы. Вещь очевидная, но мы все же ее озвучим специально: при синхронном вызове метода выполнение приостанавливается до возврата результатов с web-службы, а при асинхронном методе сразу после вызова управление возвращается клиенту. Когда выполнение метода, вызванного асинхронным способом, завершено, среда выполнения производит обратный вызов прокси-сборки. Реализация синхронного вызова метода Add() в сгенерированной прокси-сборке будет выглядеть так:

```
[System.Web.Services.Protocols.SoapMethodAttribute("http://tempuri.org/Add",
    MessageStyle=System.Web.Services.Protocols.SoapMessageStyle.ParametersInDocument)]
public int Add(int x, int y)
{
        object[] results = this.Invoke("Add", new object[] {x, y});
        return ((int)results[0]);
}
```

Как мы видим, определение каждого метода, который нужно перенаправить на web-службу, помечается при помощи атрибута SoapMethod. У метода Add() такая же сигнатура, как у исходного метода Add() на web-службе. Это очень удобно: как только клиенту потребуется обратиться к методу на web-службе, он просто привычными средствами вызывает этот метод в прокси-сборке, а та уже передает запрос на web-службу и возвращает клиенту полученные оттуда результаты.

Скорее всего, единственное изменение, которые мы захотим внести в сгенерированный исходный файл прокси-сборки, — поместить все его содержимое в специальное пространство имен. Это можно сделать опять-таки автоматически, указав имя пространства имен при помощи параметра /n утилиты wsdl.exe.

Компилируем прокси-сборку

Прежде чем мы создадим клиента, работающего через прокси-сборку, нам необходимо эту прокси-сборку создать, то есть скомпилировать соответствующий файл. Это можно сделать при помощи Visual Studio.NET (через новый проект типа C# Code Library) или компилятора командной строки csc.exe. В любом случае не забудем добавить ссылки на сборки System.Web.Services.dlL и System.Xml.dll:

```
csc /r:system.web.services.dll /r:system.xml.dll /out:C:\CalcProxy.dll /t:library
                                                                    calcproxy.cs
```

В результате будет создана прокси-сборка в виде библиотеки типов с нашим прокси-классом (рис. 15.15).

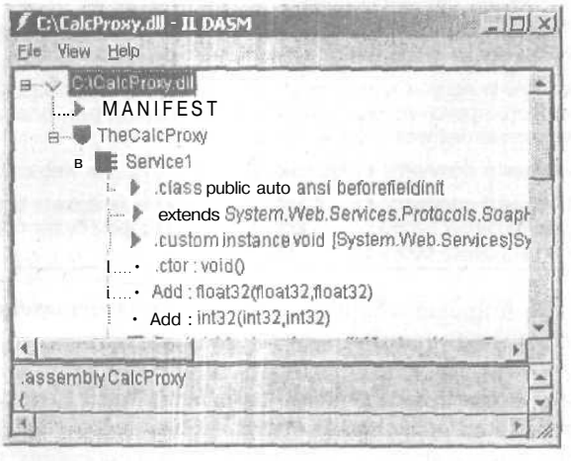


Рис. 15.15. Прокси-сборка в окне ILDasm.exe

Создание клиента для работы через прокси-сборку

Основное назначение прокси-сборки — облегчить создание клиента. И действительно, клиент, работающий через прокси-сборку, создается не просто, а очень просто. Он может быть любым — обычным клиентом Windows Forms, консольным или клиентом ASP.NET. Мы создадим самого простого клиента — консольного;

```
// Не забудьте добавить ссылку на System.Web.Services.dll!
namespace WebServiceConsumer
{
        using System;

        // Пространство имен нашей прокси-сборки
        using TheCalcProxy;

        public class WebConsumer
        {
                public static int Main(string[] args)
                {
                        // Работаем с web-службой
                        Service1 w = new Service1();
                        Console.WriteLine("100 + 100 is {0}", w.Add(100, 100));
                        try
                        {
                                w.Divide(0, 0);
                        }
                        catch(DivideByZeroException e)
                        {
                                Console.WriteLine(e.Message);
                        }
                        return 0;
                }
        }
}
```

Результат выполнения этой программы представлен на рис. 15.16. Обратите внимание, что мы смогли получить полную информацию об исключении, возникшем на web-службе.

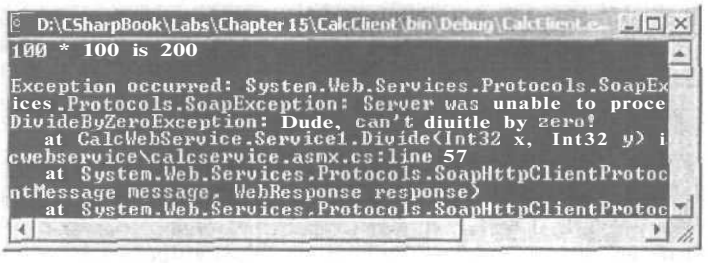


Рис. 15.16. Консольный клиент web-службы.

Код приложения CalcClient можно найти в подкаталоге Chapter 15.

Создание прокси-сборки в Visual Studio.NET

Создавать прокси-сборки можно и при помощи утилиты wsdl.exe, и прямо в Visual Studio.NET. Работа с wsdl.exe более трудоемка, но у этой утилиты есть одно существенное преимущество: мы можем указать протокол подключения, который будет использоваться прокси-сборкой (HTTP-GET, HTTP-POST или SOAP). Visual Studio.NET позволяет генерировать только прокси-сборки, работающие по протоколу SOAP. Впрочем, если мы создаем прокси-сборку, то в подавляющем большинстве случаев будем использовать для нее именно SOAP. Поэтому прокси-сборки удобнее создавать в Visual Studio.NET.

Давайте воспользуемся этой возможностью. Мы создадим очень простого клиента Windows Forms с элементарным интерфейсом, который позволит пользователю вводить два числа и передавать их четырем арифметическим методам Add(), Subtract() и т. п., которые будут перенаправлять данные через прокси-сборку на web-службу для вычислений. Создание прокси-сборки производится автоматически, нам нужно только добавить в наш проект web-ссылку (рис. 15.17).

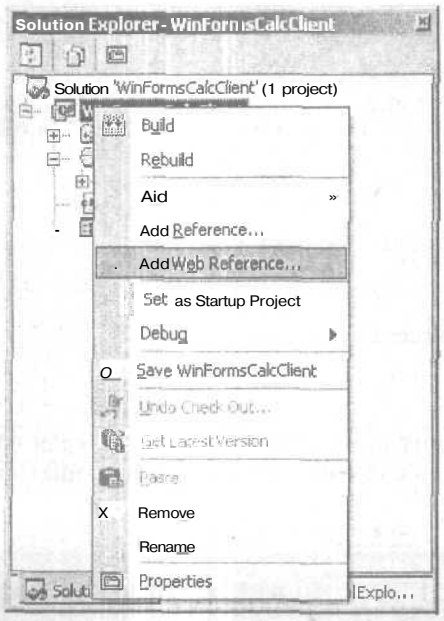


Рис. 15.17. Добавление web-ссылки

Отроется диалоговое окно Add Web Reference (Добавить web-ссылку), в котором мы сможем указать адрес URL к файлу *.asmx и просмотреть различную информацию об web-службе (рис. 15.18).

После этого выберем вкладку View Contract (Просмотреть контракт) и добавим ссылку. В окне Solution Explorer появится новый узел web-ссылки (рис. 15.19).

После этого мы уже можем напрямую работать в нашем проекте с классом прокси-сборки (он будет называться Service1). Обратите внимание, что в качестве названия пространства имен прокси-сборки выбирается имя компьютера, на котором расположена web-служба:

```
using localhost;
public class mainForm : System.Windows.Forms.Form
{
        protected void btnAdd_Click (object sender, System.EventArgs e)
        {
                localhost.Service1 w = new localhost.Service1();
                int ans = w.Add(int.Parse(txtNumb1.Text), int.Parse(txtNumb2.Text));
                lblAns.Text = ans.ToString();
        }
...
}
```

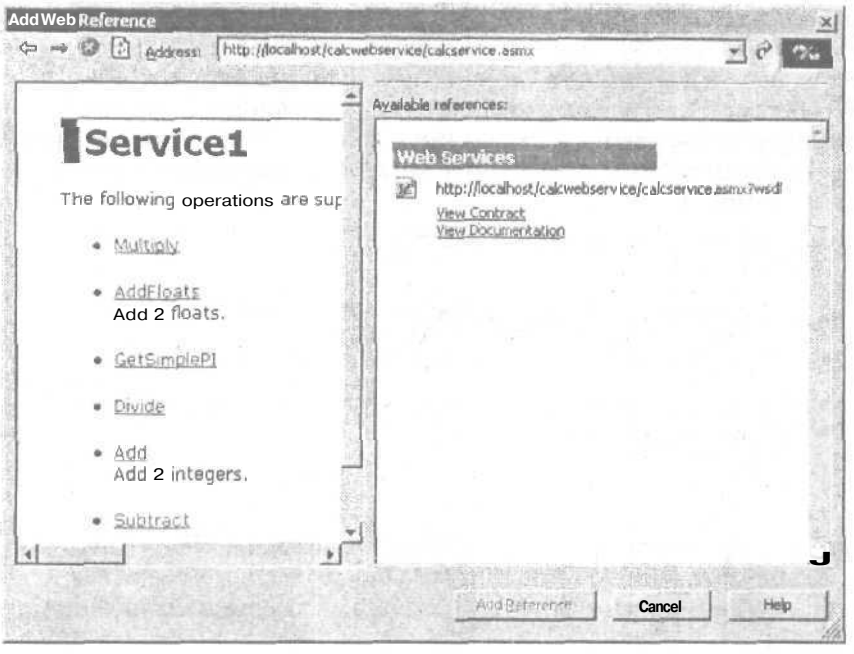


Рис. 15.18. Диалоговое окно Add Web Reference (Добавить web-ссылку)

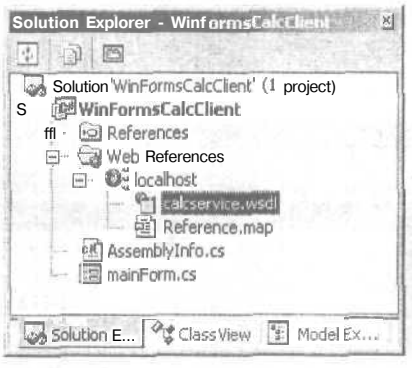


Рис. 15.19. Web-ссылка в окне Solution Explorer

Код приложения WinFormsCalcClient можно найти в подкаталоге Chapter 15.

Пример более сложной web-службы (и ее клиентов)

Однако мы немного зациклились на элементарном примере web-службы, выполняющей функции калькулятора. Возможности web-служб можно представить гораздо лучше, если реализовать методы, которые будут возвращать (конечно, толь-

ко при помощи SOAP) объекты ADO.NET DataSet, объекты пользовательских классов и массивы объектов. Созданием такой web-службы мы и займемся.

Мы начнем новый проект C# на основе шаблона Web Service и назовем его CarsWebService. Первое, что мы сделаем — создадим метод web-службы GetAllCars(), который будет возвращать объект DataSet с полным набором записей из таблицы Inventory той самой базы данных Cars, с которой мы работали в главе 13. Код для этого метода может выглядеть так:

```
// Возвращаем все записи из таблицы Inventory
public DataSet GetAllCars()
{
        // Заполняем объект DataSet данными из таблицы Inventory
        SqlConnection sqlConn = new SqlConnection();
        sqlConn.ConnectionString = "data source=.; Initial catalog=Cars;" + "user id=sa;
                                                                          password=";
        SqlDataAdapter dsc - new SqlDataAdapter("Select * from Inventory", sqlConn);
        DataSet ds = new DataSet();
        dsc.Fill(ds, "Inventory");
        return ds;
}
```

Теперь, если мы создадим клиента Windows Forms и добавим в него web-ссылку на web-службу, мы сможем вызывать метод web-службы GetAllCars(), например, для заполнения элемента управления DataGrid данными из таблицы Inventory (рис. 15.20):

```
private void mainForm_Load(object sender, System.EventArgs e)
{
        bigmanu.Service1 s = new bigmanu.Service1();
        DataSet ds - s.GetAllCars();
        dataGrid1.DataSource = ds.Tables["Inventory"];
}
```

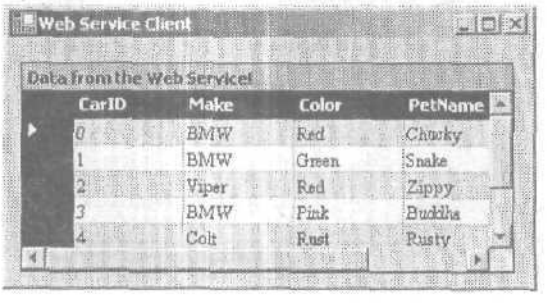


Рис. 15.20. Получение с web-службы объекта DataSet

Сериализация пользовательских типов

Протокол SOAP умеет передавать XML-представления объектов пользовательских типов с сохранением их внутреннего состояния. Чтобы убедиться в этом на примере, давайте добавим в наше приложение CarsWebService новый класс Car. Он будет очень простым: две переменные, определенные как public — для хранения информации о прозвище машины и о ее максимальной скорости, и перегруженный конструктор, который позволяет установить эти значения. Также обязательно до-

бавим одну очень важную деталь — атрибут XmlInclude (определенный в пространстве имен System.Xml.Serialization). Определение нашего класса будет выглядеть так:

```
namespace CarsWebService
{
        using System;
        using System.Xml.Serialization;

        [XmlInclude(typeof(Car))]
        public class Car
        {
                public Car(){}
                public Car(string n, int s)
                {petName = n; maxSpeed = s;}

                public string petName;
                public int maxSpeed;
        }
}
```

Процесс сохранения объекта вместе со всем внутренним состоянием (в данном случае в формате XML) называется сериализацией. Однако сериализация объекта возможна лишь в том случае, если класс этого объекта был помечен атрибутом XmlInclude.

А теперь мы определим как private массив типа ArrayList (он будет называться carList) и добавим в него несколько объектов Car — прямо в конструкторе класса нашей web-службы:

```
public Service1()
{
        InitializeComponent();

        // Добавляем объекты Car
        carList.Add(new Car("Zippy", 170));
        carList.Add(new Car("Fred", 80));
        carList.Add(new Car("Sally", 40));
}
```

После этого мы определим в web-службе два метода. GetCarList() будет возвращать весь массив carList целиком. GetACarFromList() будет возвращать конкретный объект Car по его номеру в массиве. Вот определения каждого из этих методов:

```
// Возвращаем конкретный объект Car
[WebMethod]
public Car GetACarFromList(int carToGet)
{
        if(carToGet <= carList.Count)
        {
                return (Car) carList[carToGet];
        }
        throw new IndexOutOfRangeException();
}

// Возвращаем массив объектов Car целиком
[WebMethod]
public ArrayList GetCarList()
{
        return carList;
}
```

Настраиваем клиента web-службы

Службу мы уже создали, осталось создать клиента, который к ней будет обращаться. Мы воспользуемся уже готовым клиентом из предыдущего примера (с элементом управления DataGrid). Единственное, что обязательно надо сделать, раз web-служба изменилась, — это обновить web-ссылку, например, из окна Solution Explorer. Мы добавим к графическому интерфейсу нашего клиента две кнопки (для вызова каждого из методов web-службы) и текстовое поле для ввода номера объекта Car в массиве. Код обработчика события Click для кнопки, которая будет вызывать метод GetACarFromList, будет таким:

```
protected void btnGetCar_Click (object sender, System.EventArgs e)
{
        try
        {
                bigmanu.Car c;
                bigmanu.Service1 cws = new bigmanu.Service1();
                c = cws.GetACarFromList(int.Parse(txtCarToGet.Text));
                MessageBox.Show(c.petName, "Car " + txtCarToGet.Text + " is named:");
                cws.Dispose();
        }
        catch
        {
                MessageBox.Show("No car with that number...");
        }
}
```

Результат запуска нашего клиента представлен на рис. 15.21. Помните, что мы извлекаем информацию об объекте Car из массива, а не из таблицы Inventory, которая отображается в DataGrid!

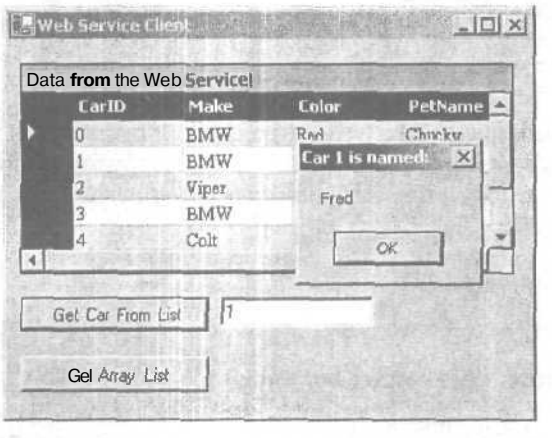


Рис. 15.21. Получаем объект Car из массива ArrayList на web-службе

Обработчик события Click для второй кнопки будет выглядеть так:

```
private void btnArrayList_Click(object sender, System.EventArgs e)
t
        bigmanu.Service1 cws = new bigmanu.ServIceH):
        object[] objs - cws.GetCarList();
```

```
        string petNames = "";

        // Переносим все данные из массива в строковую переменную
        for(int i=0; i < objs.Length; i++)
        {
                bibmanu.Car c = (bigmanu.Car)objs[i];
                petNames += c.petName + "\n";
        }
        MessageBox.Show(petNames, "Pet names for cars in array list:");
        cws.Dispose();
}
```

Создание типов для сериализации (некоторые уточнения)

При сериализации объекта в формате XML обычно является принципиальным сохранение внутреннего состояния этого объекта — чтобы его можно было потом восстановить (например, на клиенте) и продолжить с ним работу. Чтобы среда выполнения могла сериализовать внутренние данные объекта, ей необходимо получить к ним доступ. Однако если мы определили внутренние переменные как private, то получить к ним доступ среда выполнения не сможет. Как мы помним, в нашем примере мы предусмотрительно определили обе переменные как publ c:

```
[XmlInclude(typeof(Car))]
public class Car
{
        public string petName;
        public int maxSpeed;
}
```

Предположим, что мы определили их как private (как положено с точки зрения культуры программирования):

```
[XmlInclude(typeof(Car))]
public class Car
{
        public Car(){}
        public Car(string n, int s)
        { petName = n; maxSpeed = s; }

        // Попробуем меня сериализовать?
        private string petName;
        private int maxSpeed;
}
```

Если мы после этого вызовем метод GetCarList(), то в массиве также обнаружатся три объекта Car. Однако ни для одного из этих объектов не сохранится информация о внутреннем состоянии! Проблема решается просто: надо либо определять переменные как public, либо создать свойства для доступа к переменным, определенным как private:

```
[XmlInclude(typeof(Car))]
public class Car
{
...
```

```
        private string petName;
        private  int  maxSpeed;
        public string PetName
        {
                get { return petName; }
                set { petName = value; }
        }

        public int MaxSpeed
        {
                get { return maxSpeed; }
                set { maxSpeed = value; }
        }
}
```

В принципе, создание объектов, к которым можно будет обратиться через web-службу, не сильно отличается от создания обычных объектов C#. Главное — не забыть пометить такие объекты атрибутом `[XmlInclude]` и обеспечить возможность доступа к данным, определенным как `private`.

Файл Cars Web Service.asmx.cs можно найти в подкаталоге Chapter 15.

Протокол обнаружения web-службы

Последнее, о чем пойдет речь в этой главе — о службе обнаружения (discovery service) для web-службы и о файлах *.disco, которые используются для ее настройки.

Когда клиент обращается к web-службе, первое, что он должен сделать — убедиться, что web-служба по данному адресу существует. В принципе это можно сделать программным образом — в библиотеке базовых классов .NET предусмотрены для этого соответствующие типы. Однако стандартное средство сделать это — использовать службу обнаружения. Кроме того, служба обнаружения также необходима многим средствам разработки (например, она используется при добавлении web-ссылки в проект C#).

Информация о всех web-службах в конкретном виртуальном каталоге и его подкаталогах хранится в файле *.disco. Этот файл создается Visual Studio.NET полностью автоматически. Например, при создании нового проекта на основе шаблона Web Service исходный код файла *.disco будет таким:

```
<?xml version="1.0" ?>
<dynamicDiscovery xmlns="urn:schemas-dynamicdiscovery:disco.2000-03-17">
<exclude path="_vti_cnf" />
<exclude path="_vti_pvt" />
<exclude path="_vti_log" />
<exclude path="_vti_script" />
<exclude path="_vti_txt" />
<exclude path="Web References" />
</dynamicDiscovery>
```

Тег `<dynamicDiscovery>` определяет, что в ответ на запрос к службе обнаружения данный файл *.disco должен быть обработан средой выполнения ASP.NET на сервере и клиенту должен быть возвращен ответ на его запрос (ответ, конечно, будет в формате XML), сгенерированный на основе файла *.disco. Например, если мы обратимся из Internet Explorer к файлу *.disco для нашего CarsWebService, то среда выполнения ASP.NET сгенерирует нам ответ в формате XML, который представлен на рис. 15.22.

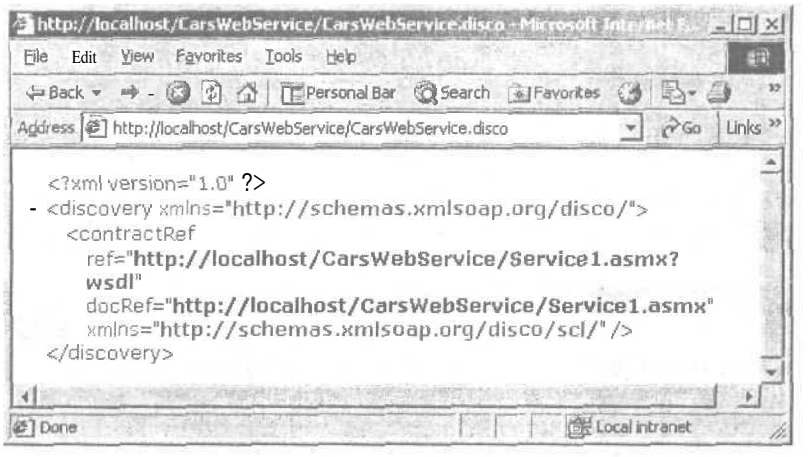


Рис. 15.22. Информация о web-службах нашего виртуального каталога

Добавление новой web-службы

Пока в нашем виртуальном каталоге есть только одна web-служба, к которой обращаются по адресу /CarsWebService/Service1.asmx. Поэтому содержимое файла *.disco вполне очевидно. Однако что произойдет, если мы добавим в наш виртуальный каталог новую web-службу? Давайте так и сделаем.

Откроем в Visual Studio.NET наш проект CarWebService и добавим в него нов™ web-службу. Проще всего это сделать при помощи меню Project (Проект) ▸ Add Web Service (Добавить web-службу). Новую web-службу мы назовем MotorBikes.asmx (рис. 15.23).

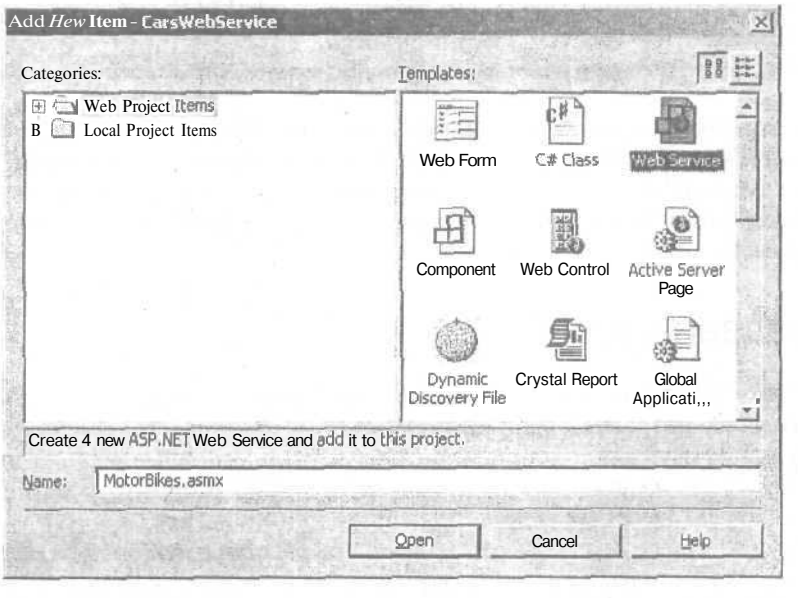


Рис. 15.23. В одном проекте Web Service может быть несколько web-служб

В классе MotorBikes будет определен единственный метод:

```
[WebMethod]
public string GetBikerDesc()
{
        return "Name: Tiny. Wight: 374 pounds.";
}
```

После перекомпиляции нашего проекта файл *.disco вернет нам информацию уже о двух web-службах (рис. 15.24).

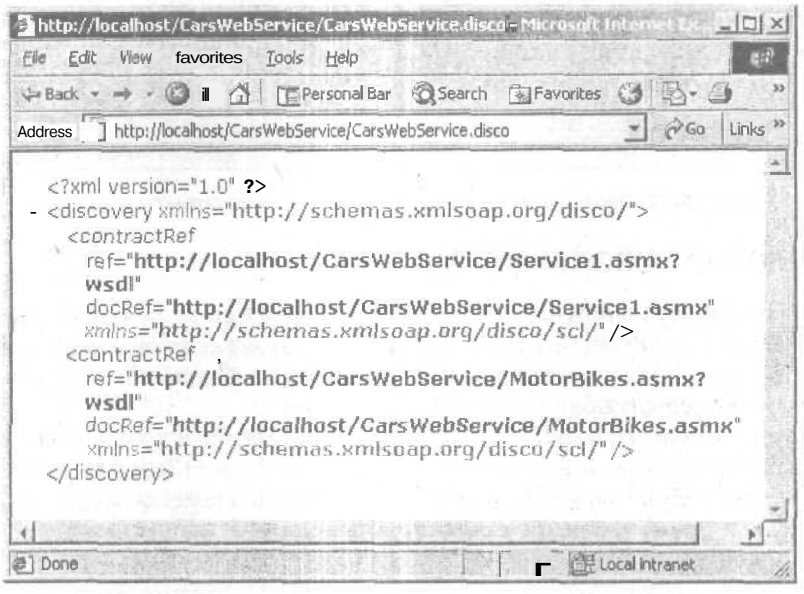


Рис. 15.24. Файл *.disco возвращает информацию обо всех web-службах виртуального каталога

Конечно, просмотр необработанного кода XML в браузере — это не самый лучший способ получения сведений о web-службах. В библиотеке базовых классов .NET предусмотрены типы, которые позволяют работать с данными, возвращаемыми службой обнаружения. Кроме того, эта информация необходима мастеру Add Web Reference и многим средствам разработки.

Подведение итогов

В этой главе были рассмотрены основные принципы и приемы построения web-служб. Вначале мы рассмотрели наиболее важные пространства имен и типы, которые используются для создания web-служб, а также основные технологии: службы обнаружения, язык описания web-служб (WSDL) и протокол подключения (HTTP-GET, HTTP-POST и SOAP).

После того как web-служба создана и мы определили в ней доступные для внешних клиентов методы при помощи атрибута [WebMethod], проще всего реализовывать клиентов таким образом, чтобы они обращались не к web-службе напрямую,

а к промежуточной прокси-сборке, которая будет перенаправлять вызовы методов на web-службу. Прокси-сборку можно сгенерировать при помощи утилиты wsdl.exe (для использования любого протокола подключения), либо из Visual Studio.NET (только для SOAP). В самом конце были рассмотрены вопросы передачи объектов пользовательских типов посредством SOAP и определения классов как пригодных для сериализации в формате XML при помощи атрибута XmlInclude.

Алфавитный указатель

Символы

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

Ж

З

И

К

M

H

Р

С

T

ЭндрюТроелсен

C# и платформа .NET. Библиотека программиста

Перевел с английского Р. Михеев

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*А. Васильев*
Технический редактор	*М. Жданова*
Литературный редактор	*А. Телов*
Художники	*Н. Биржаков, А. Келле-Пелле*
Корректоры	*В. Листова, В. Листова*
Верстка	*А. Келле-Пелле*

Лицензия ИД № 05784 от 07.09.01.
Подписано в печать 15.12.03. Формат 70×100/16.
Усл. п. л. 64,5. Доп. тираж 3000 экз. Заказ № 1310.
ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Налоговая льгота — общероссийский классификатор
продукции ОК 005-93, том 2; 953005 — литература учебная.
Отпечатано с диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания
и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

КЛУБ ПРОФЕССИОНАЛ

В 1997 году по инициативе генерального директора Издательского дома «Питер» Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан «Книжный клуб Профессионал». Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области.

Как и все развивающиеся проекты, с течением времени книжный клуб вырос в **«Клуб Профессионал»**. Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в «Клуб Профессионал» вам необходимо:

- ознакомиться с правилами вступления в «Клуб Профессионал» на страницах журнала или на сайте www.piter.com;
- выразить свое желание вступить в «Клуб Профессионал» по электронной почте postbook@piter.com или по тел. **(812) 103-73-74**;
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект «Библиотека профессионала».

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись на «Библиотеку профессионала». Она для тех, кто экономит не только время, но и деньги. Покупая комплект - книжную полку «Библиотека профессионала», вы получаете:

- скидку **15%** от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов - дополнительную скидку 3%;
- членство в «Клубе Профессионал»;
- подарок - журнал «Клуб Профессионал».

Закажите бесплатный журнал
«Клуб Профессионал».

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- по телефону; (812) 103-73-74;
- по электронному адресу: **postbook@piter.com;**
- на нашем сервере: www.piter.com;
- по почте: 197198, **Санкт-Петербург, а/я 619 ЗАО «Питер Пост».**

ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ И ОПЛАТЫ ИЗДАНИЙ:

- Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (но **без учета авиатарифа).** Книги будут высланы нашей службой «Книга-почтой» в течение двух недель после получения заказа или выхода книги из печати.
- Оплата наличными при курьерской доставке **(для жителей Москвы и Санкт-Петербурга).** Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

Вы можете заказать бесплатный журнал «Клуб Профессионал».

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM