**Эндрю Троелсен ЯЗЫК ПРОГРАММИРОВАНИЯ C# 2005 И ПЛАТФОРМА .NET 2.0 3-е издание**

**ЧАСТЬ I. Общие сведения о языке C# и платформе .NET**

**ГЛАВА 1. Философия .NET**

Каждые несколько лет программист должен быть готов кардинально обновлять свои знания, чтобы идти в ногу с новыми технологиями. Языки (C++, Visual Basic 6.0, Java), каркасы приложений (MFC, ATL, STL) и архитектуры (COM, CORBA, EJB), которые сегодня составляют "золотой фонд" разработки программного обеспечения, в будущем непременно уступят место чему-то более совершенному или, по крайней мере, более новому. Несмотря на разочарование, которое вы можете ощущать при обновлении своей базы знаний, это неизбежно. Платформа .NET - это сегодняшнее предложение Microsoft в области разработки программного обеспечения.

Целью этой главы является построение концептуального фундамента, необходимого для успешного освоения всего остального материала книги. Слава начинается с обсуждения ряда вопросов .NET, относящихся к высокому уровню, – таких как компоновочные блоки, CIL (общий промежуточный язык) и JIT-компиляция (just-in-time – точно к нужному моменту). Вдобавок к вводному обзору некоторых ключевых возможностей языка программирования C#, будет также обозначена взаимосвязь между различными элементами каркаса .NET, такими как CLR (общая языковая среда выполнения), CTS (общая система типов) и CLS (общие спецификации языка). Как вы вправе ожидать, эти темы будут исследоваться более подробно в других частях книги.

Эта глава также содержит обзор возможностей, предлагаемых библиотеками базовых классов .NET, для обозначения которых иногда используют аббревиатуру BCL (Base Class Libraries – библиотеки базовых классов) или, как альтернативу, FCL (Framework Class Libraries – библиотеки каркасных классов). Наконец, в главе обсуждается независимая от языков и платформ сущность платформы .NET (это действительно так - .NET не замыкается на операционной системе Windows).

**Предыдущее состояние дел**

Перед рассмотрением специфики .NET будет полезно рассмотреть некоторые проблемы, стимулировавшие появление предлагаемой сегодня платформы Microsoft. Чтобы получить соответствующее представление, давайте начнем эту главу с краткого урока истории, чтобы напомнить об истоках и понять ограничения, существовавшие в прошлом (в конце концов, признание существования проблемы является первым шагом на пути ее решения). После этого мы обратим наше внимание на многочисленные преимущества, которые обеспечиваются языком C# и платформой .NET.

**Подход C/Win32 API**

Традиционно разработка программного обеспечения для операционных систем семейства Windows предполагает использование языка программирования C в сочетании с Windows API (Application Programming Interface – интерфейс программирования приложений). Несмотря на тот факт, что в рамках этого проверенного временем подхода было создано очень много вполне успешных приложений, мало кто станет оспаривать то, что процесс создания приложений непосредственно с помощью API оказывается очень трудоемким делом.

Первая очевидная проблема заключается в том, что C является очень лаконичным языком. Разработчики программ на языке C вынуждены "вручную" управлять памятью, использовать безобразную арифметику указателей и ужасные синтаксические конструкции. К тому же, поскольку C является структурным языком программирования, ему не хватает преимуществ, обеспечиваемых объектно-ориентированным подходом (здесь можно вспомнить о *"макаронных" программах).*Когда вы объединяете тысячи глобальных функций и типов данных, определенных в рамках Win32 API, с языком, который и без того выглядит устрашающе, не следует удивляться тому, что среди используемых сегодня программ оказывается так много ненадежных.

**Подход C++/MFC**

Огромным шагом вперед по сравнению с подходом, предполагающим использование C/API, явился переход к применению языка программирования C++. Во многих отношениях язык C++ можно рассматривать, как объектно-ориентированную надстройку над C. Поэтому, хотя при использовании C++ уже можно использовать преимущества известных "краеугольных камней ООП" **(инкапсуляция, наследование и полиморфизм**), этот подход оставляет программиста во власти многих болезненных аспектов языка C (управление памятью "вручную", безобразная арифметика указателей и ужасные синтаксические конструкции).

Несмотря на сложность, сегодня существует множество каркасов программирования на C++. Например, MFC (Microsoft Foundation Classes – библиотека базовых классов Microsoft) снабжает разработчика набором C++-классов, упрощающих создание Win32-приложений. Главной задачей MFC является представление "разумного подмножества" Win32 API в виде набора классов, "магических" макросов и средств автоматического генерирования программного кода (обычно называемых мастерами). Несмотря на очевидную пользу указанного каркаса приложений (как и многих других средств разработчика, использующих C++), программирование на C++ остается трудной задачей, и на этом пути нелегко полностью избежать ошибок ввиду "тяжелой наследственности", обусловленной связью с языком C.

**Подход Visual Basic 6.0**

Благодаря искреннему желанию насладиться более простой жизнью, многие программисты ушли от "мира каркасов" приложений на базе C(++) к более дружественным языкам, таким, как, например, Visual Basic 6.0 (VB6). Язык VB6 стал популярным благодаря тому, что он дает возможность строить сложные интерфейсы пользователя, библиотеки программного кода (например, COM-серверы) и системы доступа к данным, затрачивая минимум усилий. В сравнении с MFC, VB6 еще глубже скрывает от глаз разработчика сложность Win32 API, используя для этого целый ряд интегрированных мастеров, внутренних типов данных, классов и специфических для VB функций.

Главным недостатком VB6 (который был устранен в Visual Basic .NET) является то, что VB6 является, скорее, "объектно-осведомленным" языком, а не полноценным объектно-ориентированным. Например, в VB6 программисту не позволяется связывать типы отношениями "подчиненности" (т.е. нет классического наследования) и нет внутренней поддержки конструкции параметризованных классов. Кроме того, VB6 не дает возможности строить многопоточные приложения, если только вы не готовы "спуститься" до низкоуровневых вызовов Win32 API (что в лучшем случае достаточно сложно, а в худшем – опасно).

**Подход Java/J2EE**

Было предложено использовать Java. Язык программирования Java является (почти) полностью объектно-ориентированным и имеет синтаксические корни в C++. Многие знают, что поддержка межплатформенной независимости – далеко не единственное преимущество Java. Java (как язык) избавлен от многих синтаксических несообразностей C++. Java (как платформа) предлагает программисту большое число встроенных "пакетов", содержащих различные определения типов. С помощью этих типов, используя "только Java", можно строить приложения, предлагающие сложный интерфейс пользователя, обеспечивающие связь с базами данных, обмен сообщениями или работу клиента в Web.

Хотя Java – очень элегантный язык, его потенциальной проблемой является то, что использование Java в цикле разработки обычно означает необходимость использования Java и для взаимодействия клиента с сервером. Как следствие, Java не позволяет возлагать большой надежды на возможности языковой интеграции, так как это идет вразрез с декларируемой целью Java (единый язык программирования для всех задач). Реальность, однако, такова, что в мире существуют миллионы строк программного кода, которым бы идеально подошло взаимодействие с более новым программным кодом. К сожалению, Java делает эту задачу проблематичной.

Java в чистом виде просто не подходит для многих приложений, интенсивно использующих графику или сложные вычисления (в этих случаях скорость работы Java оставляет желать лучшего). Для таких программ в соответствующем месте выгоднее использовать язык более низкого уровня (например, C++). Увы, пока Java не обеспечивает более широкие возможности доступа к "чужеродным" API, истинная интеграция различных языков оказывается практически невозможной.

**Подход COM**

Модель COM (Component Object Model – модель компонентных объектов) была предыдущим каркасом разработки приложений Microsoft. По сути, COM – это архитектура, "заявившая" следующее: *если класс будет построен в соответствии с правилами COM, то получится блок двоичного кода многократного использования.*

Прелесть двоичного COM-сервера в том, что способ доступа к нему не зависит от языка. Поэтому программисты, использующие C++, могут строить COM-классы, пригодные для использования в VB6. Программисты, применяющие Delphi, могут использовать COM-классы, построенные с помощью C, и т.д. Однако, и вы, возможно, об этом знаете, независимость COM от языка несколько ограничена. Например, нет возможности получить новый COM-класс из уже существующего (поскольку COM не предлагает поддержки классического наследования). Вместо этого для использования типов COM-класса вам придется указать несколько неуклюжее отношение "обладания".

Еще одним преимуществом COM является прозрачность дислокации. Используя такие конструкции, как идентификаторы приложения (AppID), "заглушки" и "заместители" в среде выполнения COM, программист может избежать необходимости непосредственного обращения к сокетам, RPC-вызовам и другими низкоуровневым элементам. Рассмотрим, например, следующий программный код VB6 COM-клиента.

' Этот блок программного кода VB6 может активизировать COM-класс,

' созданный на любом языке, поддерживающем COM, и размещенный

' в любой точке сети (включая вашу локальную машину).

Dim с as MyCOMClass

Set с = New MyCOMClass ' Размещение выясняется с помощью AppID.

с.DoSomeWork

Хотя COM можно считать очень успешной объектной моделью, внутренне она чрезвычайно сложна (по крайней мере, пока вы не потратите несколько месяцев на изучение ее внутренних механизмов – особенно если вы программируете на C++). С целью упрощения процесса разработки бинарных COM-объектов было создано множество каркасов разработки приложений с поддержкой COM. Среди них, например, библиотека ATL (Active Template Library – библиотека активных шаблонов), которая обеспечивает еще одно множество C++-классов, шаблонов и макросов, упрощающих создание COM-типов.

Многие другие языки также в значительной степени скрывают инфраструктуру COM от глаз программиста. Однако поддержки самого языка оказывается недостаточно для того, чтобы скрыть всю сложность COM. Даже при использовании относительно простого совместимого с COM языка (например, VB6), вы все равно вынуждены бороться с "хрупкими" параметрами регистрации и многочисленными проблемами, связанными с инсталляцией приложений (в совокупности называемыми *"кошмаром DLL").*

**Подход Windows DNA**

Ко всем указанным выше сложностям еще добавляется такая мелочь, как Интернет. За последние несколько лет Microsoft добавила в свое семейство операционных систем и других продуктов множество связанных с Интернет возможностей. К сожалению, создание Web-приложений в рамках совместимой с COM архитектуры Windows DNA (Distributed interNet Applications Architecture – архитектура распределенных сетевых приложений) также оказывается очень непростым делом.

Некоторая доля этой сложности вытекает из того простого факта, что Windows DNA требует использования множества технологий и языков (ASP, HTML, XML, JavaScript, VBScript, а также COM(+) и API доступа к данным, например ADO).

Одной из проблем является то, что с синтаксической точки зрения многие из этих технологий совершенно не связаны одна с другой. Например, в JavaScript используется синтаксис, во многом подобный C, a VBScript является подмножеством VB6. COM-серверы, созданные для работы в среде выполнения COM+, по виду сильно отличаются от ASP-страниц, которые их вызывают. Результат – чрезвычайно запутанная смесь технологий.

К тому же, и это, возможно, самое важное, каждый язык и каждая технология имеют свои собственные системы типов (которые могут быть совершенно не похожими одна на другую). Например, нельзя сказать, что "int" в JavaScript и "Integer" в VB6 означают в точности одно и то же.

**Решение .NET**

Слишком много для короткого урока истории. Основным выводом является то, что жизнь программиста Windows была трудна. Каркас .NET Framework является достаточно радикальной "силовой" попыткой сделать нашу жизнь легче. Решение, предложенное .NET, предполагает "изменить все" (извините, вы не можете обвинять посыльного за такое известие). Вы поймете из дальнейшего материала книги, что .NET Framework – это совершенно новая модель для создания систем как в семействе операционных систем Windows, так и множестве операционных систем, отличных от систем Microsoft, таких как Mac OS X и различные варианты Unix/ Linux. Чтобы это продемонстрировать, вот вам краткий список некоторых базовых возможностей, обеспечиваемых .NET.

• *Полноценная возможность взаимодействия с существующим программным кодом.*Это (конечно) хорошо. Существующие бинарные COM-объекты могут комбинироваться (т.е. взаимодействовать) с более новыми бинарными .NET-объектами и наоборот. Кроме того, сервисы PInvoke (Platform Invocation Services – сервисы вызова платформ) позволяют вызывать библиотеки на базе C (включая API операционной системы) из программного кода .NET.

• *Полная и тотальная интеграция языков.*В отличие от COM, платформа .NET поддерживает межъязыковое наследование, межъязыковую обработку исключений и межъязыковую отладку.

• *Общий механизм выполнения программ для всех языков с поддержкой .NET*. Одной из особенностей этого механизма является четкий набор типов, "понятных" каждому языку.

• *Библиотека базовых классов.*Эта библиотека позволяет избежать сложностей прямого обращения к API и предлагает согласованную объектную модель, используемую всеми языками с поддержкой .NET.

• *Отсутствие детализации COM.*В собственном бинарном .NET-объекте небудет места для IClassFactory, IUnknown, IDispatch, IDL-кода и "злобных" типов данных наподобие VARIANT (BSTR, SAFEARRAY и т.д.).

• *Упрощенная модель инсталляции.*Согласно спецификациям .NET, нет необходимости регистрировать соответствующую бинарную единицу в реестре системы. К тому же .NET вполне допускает существование множества версий одного \*.dll на одной машине.

На основе информации этого списка вы, вероятно, уже сами пришли к заключению, что платформа .NET не имеет ничего общего с COM (за исключением того, что оба эти каркаса разработки приложений исходят из Microsoft). Фактически единственным способом взаимодействия типов .NET и COM оказывается использование возможностей слоя взаимодействия.

**Замечание.**Описание возможностей слоя взаимодействия .NET (включая Plnvoke) выходит за рамки этой книги. Если вам потребуется подробное освещение этого вопроса, обратитесь к моей книге *COM and .NET Interoperability*(Apress, 2002).

**Главные компоненты платформы .NET (CLR, CTS и CLS)**

Теперь, когда вы знаете о некоторых преимуществах, обеспечиваемых .NET, давайте рассмотрим три ключевых (и взаимосвязанных) компонента, которые и обеспечивают эти преимущества: CLR, CTS и CLS. С точки зрения программиста .NET может интерпретироваться как новая среда выполнения программ и всеобъемлющая библиотека базовых классов. Слой среды выполнения здесь называется *CLR*(Common Language Runtime – общеязыковая среда выполнения). Главной задачей CLR являются размещение, загрузка и управление .NET-типами по вашему указанию. Кроме того, CLR отвечает за ряд низкоуровневых вопросов, таких, как, например, управление памятью и проверка безопасности.

Другим строительным блоком платформы.NET является *CTS*(Common Type System – общая система типов). Спецификации CTS полностью описывают все возможные типы данных и программные конструкции, поддерживаемые средой выполнения, указывают, как эти элементы могут взаимодействовать друг с другом и как они представляются в формате метаданных .NET (более подробная информация о метаданных будет представлена немного позже).

Вы должны понимать, что конкретный язык, совместимый с .NET, может и не поддерживать абсолютно все возможности, определенные CTS. В связи с этим используются связанные спецификации *CLS*(Common Language Specification – общеязыковые спецификации), которые определяют подмножество общих типов и программных конструкций, понятных всем языкам программирования, совместимым с .NET. Поэтому, если создаваемые вами .NET-типы опираются только на возможности, соответствующие CLS, вы можете пребывать в уверенности, что использовать их сможет любой совместимый с .NET язык. А если вы используете типы данных или программные конструкции, выходящие за пределы CLS, вы не можете гарантировать, что с вашей библиотекой программного .NET-кода сможет взаимодействовать любой язык программирования .NET.

**Роль библиотек базовых классов**

В дополнение к спецификациям CLR и CTS/CLS, платформа .NET предлагает библиотеку базовых классов, доступную всем языкам программирования .NET. Эта библиотека базовых классов не только инкапсулирует различные примитивы, такие как потоки, файловый ввод-вывод, визуализация графики и взаимодействие с различными внешними устройствами, но и обеспечивает поддержку целого ряда сервисов, необходимых для большинства современных приложений.

Например, библиотеки базовых классов определяют типы, упрощающие доступ к базам данных, работу с XML, поддержку программной безопасности и создание Web-приложений (а также обычных настольных и консольных приложений) клиента. Схема высокоуровневых взаимосвязей между CLR, CTS, CLS и библиотекой базовых классов показана на рис. 1.1.

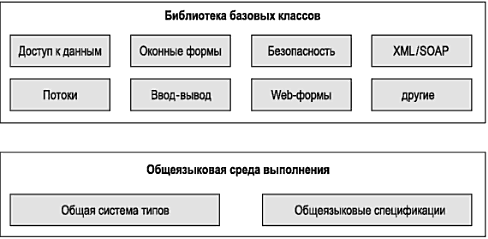


Рис. 1.1. CLR, CTS, CLS и библиотека базовых классов

**Роль языка C#**

С учетом того, что принципы .NET так радикально отличаются от предшествующих технологий, Microsoft разработала новый язык программирования, C# (произносится "си-диез"), специально для использования с этой новой платформой. Язык C# является языком программирования, по синтаксису очень похожим на Java (но не идентичным ему). Однако называть C# "переработанным" вариантом Java будет неверно. C#, как и Java, основан на синтаксических конструкциях C++. Так же, как и Java, C# можно называть "рафинированной" версией C++ – в конце концов, это языки одного семейства.

Многие синтаксические конструкции C# построены с учетом решений, принятых в Visual Basic 6.0 и C++. Например, как и в VB6, в C# поддерживаются формальные свойства типов (в противоположность традиционным методам get и set) и возможность объявления методов с переменным числом аргументов (через массивы параметров). Подобно C++, в C# позволяется перегрузка операций, а также создание структур, перечней и функций обратного вызова (посредством делегатов).

Благодаря тому, что C# является гибридом множества языков, он является продуктом, который синтаксически так же "чист", как Java (если не "чище"), почти так же прост, как VB6, и обладает почти такой же мощью и гибкостью, как C++ (без соответствующих "ужасных" конструкций). По сути, язык C# предлагает следующие возможности (многие из которых присущи и всем другим языкам программирования, обеспечивающим поддержку .NET).

• Не требуется никаких указателей! Программы на C# обычно не требуют прямого обращения к указателям (хотя имеется возможность получить к ним доступ на более низком уровне, если вы сочтете это абсолютно необходимым).

• Автоматическое управление памятью через сборку мусора. По этой причине в C# не поддерживается ключевое слово delete.

• Формальные синтаксические конструкции для перечней, структур и свойств классов.

• Аналогичная C++ перегрузка операций для пользовательских типов, но без лишних сложностей (например, вам не требуется контролировать "возвращение \*this для связывания").

• В C# 2005 имеется возможность строить общие типы и общие члены с использованием синтаксиса, очень похожего на шаблоны C++.

• Полная поддержка техники программирования, основанной на использовании интерфейсов.

• Полная поддержка технологии аспектно-ориентированного программирования (АОП) через атрибуты. Эта ветвь разработки позволяет назначать характеристики типам и их членам, чтобы уточнять их поведение.

Возможно, самым важным для правильного понимания языка C#, поставляемого Microsoft в связке с платформой .NET, является то, что получаемый с помощью C# программный код может выполняться только в среде выполнения .NET (вы не сможете использовать C# для построения "классического" COM-сервера или автономного приложения Win32 API). Официальный термин, который используется для описания программного кода, предназначенного для среды выполнения .NET, – управляемый *программный код*(managed code). Бинарный объект, содержащий такой управляемый программный код, называется*компоновочным блоком*(подробнее о компоновочных блоках мы поговорим немного позже). С другой стороны, программный код, который не может непосредственно управляться средой выполнения .NET, называется *неуправляемым программным кодом*(unmanaged code).

**Другие языки программирования с поддержкой .NET**

Вы должны понимать, что C# является не единственным языком, ориентированным на платформу .NET. Когда платформа .NET была впервые представлена общественности на Профессиональной конференции разработчиков Microsoft в 2000 году, ряд производителей объявили, что они уже разрабатывают версии соответствующих компиляторов, совместимые с .NET. На момент создания этой книги десятки различных языков подверглись влиянию .NET. В дополнение к пяти языкам, которые предлагаются в Visual Studio 2005 (C#, J#, Visual Basic .NET, Managed Extensions для C++ и JScript .NET), имеются также .NET-компиляторы для Smalltalk, COBOL и Pascal (это далеко не полный перечень).

Материал этой книги почти исключительно посвящен языку C#, но в табл. 1.1 приводится список других языков программирования, совместимых с .NET, и указано, где найти более подробную информацию о них (учтите, что соответствующие адреса URL могут измениться).

Таблица 1.1. Некоторые из языков программирования, совместимых с .NET

|  |  |
| --- | --- |
| **Адрес Web-страницы языка .NET** | **Описание** |
| [http://www.oberon.ethz.ch/oberon.NET](http://www.oberon.ethz.ch/oberon.net) | "Домашняя" страница Active Oberon .NET |
| <http://www.usafa.af.mil/df/dfcs/bios/mcc_html/a_sharp.cfm> " | Домашняя" страница А# (порт Ada для платформы .NET) |
| [http://www.netcobol.com](http://www.netcobol.com/) | Для тех, кого интересует COBOL .NET |
| [http://www.eiffel.com](http://www.eiffel.com/) | Для тех, кого интересует Eiffel .NET |
| <http://www.dataman.ro/dforth> | Для тех, кого интересует Forth .NET |
| <http://www.silverfrost.com/ll/ftn95/ftn95_fortran_95_for_windows.asp> | Для тех, кого интересует Fortran .NET |
| [http://www.vmx-net.com](http://www.vmx-net.com/) | Оказывается, доступен даже Smalltalk .NET |

Следует учесть, что информация табл. 1.1 не является исчерпывающей. Списки компиляторов для .NET имеются на многих Web-узлах, и один из таких списков должен быть на странице http://www.dotnetpowered.com/languages.aspx (опять же, точный адрес URL может измениться). Я рекомендую посетить эту страницу, поскольку вас непременно заинтересуют хотя бы некоторые из языков .NET (может, кому-то понадобится LISP .NET).

**Жизнь в многоязычном окружении**

В начале процесса осмысления разработчиком языково-агностической природы платформы .NET, у него возникает множество вопросов и прежде всего, следующий: "Если все языки .NET при компиляции преобразуются в "управляемый программный код", то почему существует не один, а множество компиляторов?". Ответить на этот вопрос можно по-разному. Во-первых, мы, программисты, бываем очень привередливы, когда дело касается выбора языка программирования (я здесь тоже не исключение). Некоторые из нас предпочитают языки с многочисленными точками с запятыми и фигурными скобками, но с минимальным набором ключевых слов. Другим нравятся языки, предлагающие более "человеческие" синтаксические лексемы (как Visual Basic .NET). А кто-то не пожелает отказываться от своего опыта работы на большой ЭВМ и захочет перенести его на платформу .NET (используя COBOL .NET).

А теперь скажите честно: если бы Microsoft предложила единственный "официальный" язык .NET, например, на базе семейства BASIC, то все ли программисты были бы рады такому выбору? Или если бы "официальный" язык .NET был основан на синтаксисе Fortran, то сколько людей в мире вообще проигнорировало бы платформу .NET? Поскольку среда выполнения .NET демонстрирует меньшую зависимость от языка, используемого для построения управляемого программного кода, программисты .NET могут, не меняя своих синтаксических предпочтений, обмениваться скомпилированными компоновочными блоками со своими коллегами, другими отделами и внешними организациями (не обращая внимания на то, какой язык .NET используется там).

Еще одно полезное преимущество интеграции различных языков .NET в одном унифицированном программном решении вытекает из того простого факта, что каждый язык программирования имеет свои сильные (а также слабые) стороны. Например, некоторые языки программирования имеют превосходную встроенную поддержку сложных математических вычислений. В других лучше реализованы финансовые или логические вычисления, взаимодействие с центральными компьютерами и т.д. Когда преимущества конкретного языка программирования объединяются с преимуществами платформы .NET, выигрывают все.

Конечно, вы можете разрабатывать программное обеспечение, не выходя за рамки своего любимого языка .NET. Но, изучив синтаксис одного языка .NET, вам будет очень легко освоить любой другой. Это тоже очень выгодно, особенно консультантам. Если вашей специализацией является C#, но вы оказались на узле клиента, который "привязан" к Visual Basic .NET, то сможете почти сразу разобрать соответствующий программный код (поверьте!), воспользовавшись указанным преимуществом .NET. На этом и остановимся.

**Компоновочные блоки .NET**

Независимо от того, какой язык .NET вы выберете для программирования, вы должны понимать, что хотя бинарные .NET-единицы имеют такие же расширения файлов, как COM-серверы и неуправляемые программы Win32 (\*.dll или \*.exe), их внутреннее устройство совершенно иное. Например, бинарные .NET-единицы \*.dll не экспортируют методы для упрощения коммуникации со средой выполнения COM (поскольку .NET – это не COM). Бинарные .NET-единицы не описываются с помощью библиотек COM-типов и не регистрируются в реестре системы. Наверное, самым важным является то, что бинарные .NET-единицы содержат не специфические для платформы инструкции, а независимые от платформы IL-инструкции (Intermediate Language – промежуточный язык) и метаданные типов. На рис. 1.2 это показано схематически.

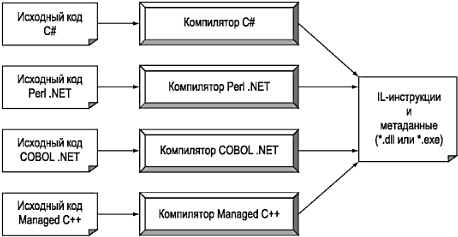


Рис. 1.2. Все .NET-компиляторы генерируют IL-инструкции и метаданные

**Замечание.**Относительно сокращения "IL" здесь уместно сказать несколько дополнительных слов. В ходе разработки .NET официальным названием для IL было Microsoft (intermediate Language (MSIL). Однако в вышедшей версии .NET это название было изменено на OIL (Common Intermediate Language – общий промежуточный язык). Поэтому вам следует знать, что в публикациях, посвященных .NET, сокращения IL, MSIL и CIL обозначают одно и то же. В соответствии с терминологией, принятой сегодня, в тексте этой книги используется сокращение CIL.

После создания \*.dll или \*.exe с помощью подходящего .NET-компилятора, соответствующий модуль упаковывается в *компоновочный блок.*Подробное описание компоновочных блоков .NET имеется в главе 11. Однако, чтобы продолжить наше обсуждение среды выполнения .NET, вы должны знать основные особенности формата этих новых файлов.

Как уже было сказано, компоновочный блок содержит программный код CIL, который концептуально напоминает байт-код Java в том смысле, что он не компилируется в специфические для соответствующей платформы инструкции, пока это не станет абсолютно необходимо. Обычно "абсолютная необходимость" означает момент, когда на какой-то блок CIL-инструкций (например, реализацию метода) выполняется ссылка для его использования в среде выполнения .NET.

В добавление к CIL-инструкциям, компоновочные блоки также содержат *метаданные,*которые подробно описывают особенности каждого "типа" внутри данной бинарной .NET-единицы. Например, если вы имеете класс с именем SportsCar, соответствующие метаданные типа будут описывать такие элементы, как базовый класс SportsCar и интерфейсы, реализуемые SportsCar (если таковые имеются), а также содержать полные описания всех членов, поддерживаемых типом SportsCar.

Метаданные .NET более совершенны по сравнению с метаданными COM. Вы, возможно, уже знаете, что бинарные COM-объекты обычно описываются с помощью библиотеки ассоциированных типов, а это почти то же самое, что и бинарная версия IDL-кода (Interface Definition Language – язык определения интерфейса). Проблема использования COM-информации в том, что эта информация не обязательна, и IDL-код не может документировать внешние серверы, которые нужны для правильного функционирования данного COM-сервера. В противоположность этому метаданные .NET присутствуют обязательно и автоматически генерируются соответствующим .NET-компилятором.

Наконец, в добавление к CIL и метаданным типов, сами компоновочные блоки также описываются с помощью метаданных, для которых используют специальное называние *манифест*(manifest). Манифест содержит информацию о текущей версии компоновочного блока, информацию о "культуре" (используемую для локализации строк и графических ресурсов) и список всех ссылок на внешние компоновочные блоки, которые требуются для правильного функционирования. Из следующих глав вы узнаете о различных инструментах, которые могут использоваться для исследования типов компоновочного блока, расшифровки его метаданных и манифеста.

**Одномодульные и многомодульные компоновочные блоки**

Во многих случаях компоновочные блоки .NET- это просто файлы двоичного кода (\*.dll или \*.exe). Поэтому, если вы строите \*.dll .NET, можно считать, что файл двоичного кода и компоновочный блок – это одно и то же. Точно также, если вы строите выполняемое приложение для настольной системы, файл \*.exe тоже можно считать компоновочным блоком. Но из главы 11 вы узнаете, что указанное соответствие не столь однозначно. Строго говоря, если компоновочный блок состоит из одного модуля \*.dll или \*.exe, вы имеете *одномодульный компоновочный блок.*Одномодульный компоновочный блок содержит весь необходимый код CIL, метаданные и манифест в одном автономном отдельном пакете.

*Многомодульные компоновочные блоки,*в свою очередь, складываются из множества бинарных .NET-единиц, каждая из которых называется *модулем.*При этом один из таких модулей (он называется*первичным модулем)*должен содержать манифест компоновочного блока (и может содержать также CIL-инструкции и метаданные различных типов). Остальные связанные модули содержат манифест уровня модуля, CIL и метаданные типов. Как вы можете догадаться, в манифесте первичного модуля компоновочного блока документируется набор необходимых "второстепенных" модулей.

Но зачем создавать многомодульные компоновочные блоки? Когда вы делите компоновочный блок на отдельные модули, вы получаете более гибкие возможности инсталляции. Например, если пользователь ссылается на удаленный компоновочный блок, то среда выполнения загрузит на его машину только необходимые модули. Поэтому вы можете сконструировать свой компоновочный блок так, чтобы редко используемые типы (например, HardDriveReformatter) были выделены в отдельный автономный модуль.

Если все ваши типы размещаются в компоновочном блоке, представляющем собой единый файл, конечному пользователю придется загружать большой набор данных, из которых в действительности могут понадобиться далеко не все (а это, очевидно, означает лишнюю трату времени). Итак, компоновочный блок на самом деле *логически сгруппирован*в один или несколько модулей, которые должны инсталлироваться и использоваться, как единое целое.

**Роль CIL**

Теперь, когда вы имеете начальное представление о компоновочных блоках .NET, давайте немного подробнее обсудим роль общего промежуточного языка (CIL). CIL- это язык, находящийся выше любого набора инструкций, специфического для конкретной платформы. Независимо от того, какой язык .NET вы выберете для использования, соответствующий компилятор сгенерирует инструкции CIL. Например, следующий программный код C# моделирует тривиальный калькулятор. Не пытаясь пока что полностью понять синтаксис этого примера, обратите внимание на формат метода Add() в классе Calc.

// Calc.cs

using System;

namespace CalculatorExample {

**// Этот класс содержит точку входа приложения.**

 public class CalcApp {

  static void Main() {

   Calc с = new Calc();

   int ans = c.Add(10, 84);

   Console.WriteLine("10 + 84 is {0}.", ans);

**// Ждать, пока пользователь не нажмет клавишу ввода.**

   Console.ReadLine();

  }

 }

**// C#-калькулятор.**

 public class Calc {

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

 }

}

После того как компилятор C# (csc.exe) скомпилирует этот файл исходного кода, вы получите состоящий из одного файла компоновочный блок \*.exe, который содержит манифест, CIL-инструкции и метаданные, описывающие каждый аспект классов Calc и CalcApp. Например, если вы откроете этот компоновочный блок с помощью ildasm.exe (мы рассмотрим ildasm.exe немного позже в этой же главе), вы увидите, что метод Add () в терминах CIL представляется так.

.method public hidebysig instance **int32 Add(int32 x, int32 y)**cil managed

{

 // Code size 8 (0x8)

 .maxstack 2

 .locals init ([0] int32 CS$l$0000)

 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

} // end of method Calc::Add

Не беспокойтесь, если вы пока не в состоянии понять CIL-код для этого метода – в главе 15 будут описаны основы языка программирования CIL. Здесь следует сконцентрироваться на том, что компилятор C# генерирует CIL-код, а не специфические для платформы инструкции.

Напомним теперь, что это верно для всех .NET-компиляторов. Для иллюстрации предположим, что вы создали аналогичное приложение с помощью Visual Basic .NET (VB .NET), а не с помощью C#.

**' Calc.vb**

Imports System

Namespace CalculatorExample

 ' **VB .NET 'Модуль' – это класс, содержащий только ' статические члены.**

 Module CalcApp

  Sub Main()

   Dim ans As Integer

   Dim с As New Calc

   ans = c.Add(10, 84)

   Console.WriteLine("10 + 84 is {0}.", ans)

   Console.ReadLine()

  End Sub

 End Module

 Class Calc

  Public Function **Add**(ByVal x As Integer, ByVal у As Integer) As Integer

   Return x + у

  End Function

 End Class

End Namespace

Если теперь проверить CIL-код для метода Add(), вы обнаружите подобные инструкции (слегка "подправленные" компилятором VB .NET).

.method public instance int32 Add(int32 x, int32 y) cil managed

{

 // Code size 9 (0x9)

 .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: br.s IL\_0007

 IL\_0007: ldloc.0

 IL\_0008: ret

} // end of method Calc::Add

**Преимущества CIL**

Вы можете спросить, зачем компилировать исходный код в CIL, а не прямо в набор специальных системных команд. Одним из преимуществ этого является интеграция языков, поскольку вы уже убедились, что все компиляторы .NET выдают приблизительно одинаковые наборы CIL-инструкций. Поэтому все языки могут взаимодействовать в рамках четко обозначенной двоичной "арены".

Кроме того, поскольку CIL демонстрирует независимость от платформы, каркас .NET Framework тоже оказывается независимым от платформы, обеспечивая то, к чему так привыкли разработчики Java (единую базу программного кода, способного работать во многих операционных системах). Фактически уже имеется международный стандарт для языка C#, а значительная часть платформы .NET реализована для множества операционных систем, отличных от Windows (более подробная информация об этом имеется в конце главы). Но, в отличие от Java, .NET позволяет строить приложения, используя язык вашего предпочтения.

**Преобразование CIL-кода в набор инструкций, соответствующих платформе**

Ввиду того, что компоновочные блоки содержат CIL-инструкции, а не инструкции для конкретной платформы, программный код CIL перед использованием приходится в фоновом режиме компилировать. Объект, который компилирует программный код CIL в инструкции, понятные процессору машины, называется *JIT-компилятором*(just-in-time – точно к нужному моменту), который иногда "по-дружески" также называют Jitter. Среда выполнения .NET использует JIT-компилятор, соответствующий конкретному процессору и оптимизированный для соответствующей платформы.

Например, если ваше .NET-приложение предназначено для выполнения на "компактном" устройстве (таком, как, например, КПК), то соответствующий JIT-компилятор будет иметь специальные средства для учета условий ограниченности памяти. Если это компоновочный блок для серверной системы (где объем памяти редко оказывается проблемой), то соответствующий JIT-компилятор будет оптимизирован для работы в условиях достаточного объема памяти. Таким образом разработчики получают возможность создавать только один блок программного кода, который с помощью JIT-компиляции можно выполнять на машинах с разной архитектурой.

К тому же, при компиляции CIL-инструкций в соответствующий машинный код JIT-компилятор поместит результаты компиляции в кэш в соответствии с тем, как этого требует соответствующая операционная система. Так, при первом вызове метода с именем PrintDocument() соответствующие CIL-инструкции компилируются в конкретные инструкции платформы и сохраняются в памяти для использования в дальнейшем. Поэтому при следующих вызовах PrintDocument () необходимости в повторной компиляции CIL не возникает.

**Роль метаданных типов .NET**

Кроме CIL-инструкций, компоновочный блок .NET содержит исчерпывающие и точные метаданные, описывающие все его типы (классы, структуры, перечни и т.д.), определенные в бинарном объекте, и все члены каждого типа (свойства, методы, события и т.д.). К счастью, задача создания метаданных всегда возлагается на компилятор (а не на программиста). По причине того, что метаданные .NET так подробны и точны, компоновочные блоки оказываются единицами, способными себя полностью описать, – настолько полно, что для бинарных .NET-объектов не возникает необходимости регистрироваться в реестре системы.

Для иллюстрации формата метаданных типов .NET давайте рассмотрим метаданные, сгенерированные для метода Add() C#-класса Calc, представленного выше (метаданные, генерируемые для VB .NET-версии метода Add(), оказываются аналогичными).

TypeDef #2 (02000003)

-----------------------------------------------------------

 TypDefName: CalculatorExample.Calc (02000003)

 Flags: [Public] [AutoLayout] [Class] [AnsiClass] [BeforeFieldlnit] (00100001)

 Extends: 01000001 [TypeRef] System.Object

 Method #1 (06000003)

-----------------------------------------------------------

 MethodName: Add (06000003)

 Flags: [Public] [HideBySig] [ReuseSlot] (00000086)

 RVA: 0x00002090

 ImplFlags: [IL] [Managed] (00000000)

 CallCnvntn: [DEFAULT]

 hasThis

 ReturnType: I**4**

**2 Arguments**

  Argument #1: I4

  Argument #2: I4

**2 Parameters**

  (1) ParamToken: (08000001) Name: x flags: [none] (00000000)

  (2) ParamToken: (08000002) Name: у flags: [none] (00000000)

Метаданные используются средой выполнения .NET, а также различными средствами разработки. Например, возможность IntelliSense, предлагаемая в Visual Studio 2005 в режиме проектирования, основана на чтении метаданных компоновочного блока. Метаданные используются различными утилитами просмотра объектов, инструментами отладки и самим компилятором C#. Для полноты картины заметим также, что использование метаданных лежит в основе множества .NET-технологий, включая удаленный доступ, отображение типов, динамическое связывание, Web-сервисы XML и сериализацию объектов.

**Роль манифеста компоновочного блока**

Наконец вспомним, что компоновочный блок .NET содержит также метаданные, описывающие сам компоновочный блок (эти метаданные называются *манифест*)*.*Среди всего прочего, в манифесте документируются все внешние компоновочные блоки, которые требуются текущему компоновочному блоку для корректного функционирования, указан номер версии компоновочного блока, информация об авторских правах и т.д. Подобно метаданным типов, генерирование манифеста компоновочного блока тоже является задачей компилятора. Вот некоторые подходящие для иллюстрации элементы манифеста CSharpCalculator.exe.

.assembly extern mscorlib

{

 .publickeytoken = (B7 7A 5C 56 19 34 E0 89)

 .ver 2:0:0:0

}

.assembly CSharpCalculator

{

 .hash algorithm 0x00008004

 .ver 0:0:0:0

}

.module CSharpCalculator.exe

.imagebase 0x00400000

.subsystem 0x00000003

.file alignment 512

.corflags 0x00000001

По сути, этот манифест содержит указания на внешние компоновочные блоки, необходимые для CSharpCalculator.exe (для этого используется директива.assembly extern), а также различные характеристики самого компоновочного блока (номер версии, имя модуля и т.д.).

**Общая система типов**

Компоновочный блок может содержать любое число четко определенных "типов". В мире .NET "тип" – это просто общий термин, используемый для обозначения любого элемента из множества *{класс, структура, интерфейс, перечень, делегат}.*При построении решений с помощью любого языка .NET вы, скорее всего, будете взаимодействовать с каждым из этих типов. Например, компоновочный блок может определять один класс, в котором реализовано ряд интерфейсов. И, возможно, один из методов интерфейса будет принимать перечень в качестве входного параметра, а возвращать некоторую структуру.

Напомним, что CTS (общая система типов) – это формальное описание того, как должны определяться типы, подходящие для использования в среде CLR. Обычно внутренние механизмы CTS важны только тем, кто создает средства разработки и/или строит компиляторы для платформы .NET. Но для любого программиста .NET важно знать, как работать с пятью типами, определяемыми спецификациями CTS для выбранного разработчиком языка программирования. Ниже предлагается краткий обзор соответствующих вопросов.

**Тип класса**

Любой язык, совместимый с .NET, поддерживает, как минимум, тип *класса,*который является "краеугольным камнем" объектно-ориентированного программирования (ООП). Класс может состоять из любого числа членов (таких, как свойства, методы и события) и элементов данных (таких, как поля). В C# классы объявляются с помощью ключевого слова class.

**// Тип класса C#.**

public class Calc {

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

}

Процесс построения типов класса CTS в C# будет рассматриваться в главе 4, но ряд общих характеристик типов класса приводится в табл. 1.2.

**Таблица 1.2.**Характеристики классов CTS

|  |  |
| --- | --- |
| **Характеристика класса** | **Описание** |
| Изолированность | *Изолированные*классы не могут быть базой для создания других классов, т.е. не позволяют наследование |
| Наличие интерфейсов | *Интерфейс -*это набор абстрактных членов, обеспечивающих взаимодействие между объектом и пользователем этого объекта. Спецификации CTS не ограничивают число интерфейсов, реализуемых в рамках класса |
| Абстрактность или конкретность | *Абстрактные*классы не позволяют непосредственное сознание их экземпляров – они предназначены для определения общих элементов поведения производных типов. *Конкретные*классы могут быть созданы непосредственно |
| Видимость | Каждый класс должен иметь атрибут видимости (visibility). По сути, этот атрибут указывает, доступен данный класс для использования внешними компоновочными блоками или он доступен для использования только внутри определяющего этот класс компоновочного блока (как, например, приватный класс справки) |

**Тип структуры**

Понятие структуры в CTS также формализовано. Если вы знаете C, вам будет приятно узнать, что эти пользовательские типы "выжили" и в мире .NET (хотя внутренне они ведут себя немного по-иному). Упрощенно говоря, *структура*- это "облегченный" тип класса с семантикой на базе значений. Более подробная информация о структурах предлагается в главе 3. Обычно структуры лучше всего подходят для моделирования геометрических и математических данных, и в C# для создания структур используется ключевое слово struct.

**// Тип структуры C#.**

struct Point {

**// Структуры могут содержать поля.**

 public int xPos, yPos;

**// Структуры могут содержать параметризованные конструкторы.**

 public Point (int x, int у) {xPos = x; yPos = y;}

**// Структуры могут определять методы.**

  public void Display() {

  Console.WriteLine("({0}, {1})", xPos, yPos);

 }

}

**Тип интерфейса**

*Интерфейс*- это именованная коллекция определений абстрактных членов, которая может поддерживаться (т.е. реализоваться) данным классом или структурой. В отличие от модели COM, интерфейсы .NET *не являются*производными одного общего базового интерфейса, такого как IUnknown. В C# типы интерфейса определяются с помощью ключевого слова interface, например:

**// Тип интерфейса C#.**

public interface IDraw {

 void Draw ();

}

Сами по себе интерфейсы не очень полезны. Однако, когда класс или структура реализуют данный интерфейс своим собственным уникальным образом, вы можете запросить доступ к соответствующим функциональным возможностям, используя ссылку на интерфейс в полиморфной форме. Программирование на базе интерфейсов будет рассматриваться в главе 7.

**Тип перечня**

*Перечень*- это удобная программная конструкция, в которой группируются пары "имя-значение". Предположим, вы создаете видеоигру, в которой игроку позволяется выбрать персонажа в одной из трех категорий: Wizard (маг), Fighter (воин) или Thief (мошенник). Вместо того чтобы использовать и отслеживать числовые значения, соответствующие каждой из возможностей, вы можете построить перечень, используя для этого ключевое слово enum.

**// Тип перечня C#.**

public enum CharacterType {

 Wizard = 100,

 Fighter = 200,

 Thief = 300

}

По умолчанию для каждого элемента выделяется блок памяти, соответствующий 32-битовому целому, но при необходимости это значение можно изменить (например, в случае программирования для устройств с малыми объемами памяти, таких как КПК). Спецификации CTS предполагают, что типы перечня должны "получаться" из общего базового класса, System.Enum. Из главы 3 вы узнаете, что этот базовый класс определяет ряд весьма полезных членов, которые позволяют программно извлекать, обрабатывать и преобразовывать соответствующие пары "имя-значение".

**Тип делегата**

*Делегат*- это .NET-эквивалент обеспечивающих типовую безопасность указателей функций C. Главное отличие заключается в том, что делегат .NET- это *класс,*получаемый путем наследования System.MulticastDelegate, а не просто указатель на конкретный адрес в памяти. В C# делегаты объявляются с помощью ключевого слова delegate.

**// Этот тип делегата C# может 'указывать' на любой метод, возвращающий**

**// целое значение и получающий на вход два целых значения.**

public delegate int BinaryOp(int x, int y);

Делегаты полезны, когда требуется обеспечить элементу возможность передачи вызова другому элементу, что создает основу для архитектуры обработки событий .NET. В главах 8 и 14 будет показано, что делегаты имеют внутреннюю поддержку методов многоадресного (предназначенного для множества получателей) и асинхронного вызова.

**Члены типов**

Теперь после рассмотрения всех типов, имеющих формальное определение в CTS, вы должны осознать, что большинство типов может иметь любое число *членов.*Формально *член типа*- это любой элемент множества *{конструктор, деструктор*(finalizer), *статический конструктор, вложенный тип, операция, метод, свойство, индексатор, поле, поле только для чтения, константа, событие}.*

Спецификации CTS определяют различные "характеристики", которые могут связываться с данным членом. Например, каждый член имеет признак, характеризующий его доступность (открытый, частный, защищенный и т.д.). Некоторые члены могут объявляться как абстрактные, чтобы навязать полиморфное поведение производным типам, или виртуальные, чтобы определить фиксированную (но допускающую замену) реализацию. Большинство членов может обозначаться как статические ("привязанные" к уровню класса) или как члены экземпляра ("привязанные" к уровню объекта). Конструкция членов типа будет подробно обсуждаться в следующих главах.

**Замечание.**Как будет показано в главе 10, в .NET 2.0 поддерживается конструкция обобщенных типов и обобщенных членов.

**Встроенные типы данных CTS**

Еще одной особенностью CTS, о которой следует знать, является то, что спецификации CTS определяют четкий набор базовых типов данных. Хотя каждый язык обычно предлагает свое уникальное ключевое слово, используемое для объявления конкретного встроенного типа данных CTS, все эти ключевые слова в конечном счете приводят к соответствующему типу, определенному в компоновочном блоке mscorlib.dll. Взгляните на табл. 1.3, предлагающую информацию о том, как базовые типы данных CTS выражены в разных языках .NET.

Таблица 1.3. Встроенные типы данных CTS

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип данных CTS** | **Ключевое слово VB .NET** | **Ключевое слово C#** | **Ключевое слово Managed Extensions for C++** |
| System.ByteByte | Byte | byte | unsigned char |
| System.SByteSByte | SByte | sbyte | signed char |
| System.Int16 | Short | short | short |
| System.Int32 | Integer | int | int или long |
| System.Int64 | Long | long | \_\_int64 |
| System.UInt16 | UShort | ushort | unsigned short |
| System.UInt32 | UInteger | uint | unsigned int или unsigned long |
| System.UInt64 | ULong | ulong | unsigned \_\_int64 |
| System.SingleSingle | Single | float | Float |
| System.DoubleDouble | Double | double | Double |
| System.ObjectObject | Object | object | Object^ |
| System.CharChar | Char | char | wchar\_t |
| System.StringString | String | String | String^ |
| System.DecimalDecimal | Decimal | decimal | Decimal |
| System.BooleanBoolean | Boolean | bool | Bool |

**Общеязыковые спецификации**

Вы, конечно, знаете, что разные языки программирования выражают одни и те же программные конструкции в своих уникальных терминах. Например, в C# конкатенация строк обозначается знаком "плюс" (+), a в VB .NET для этого используется амперсанд (&). Даже тогда, когда два языка выражают одну и ту же программную идиому (например, функцию, не возвращающую никакого значения), весьма вероятно то, что при этом используется разный синтаксис.

**' Не возвращающий ничего метод VB.NET.**

Public Sub MyMethod()

**' Некоторый программный код…**

End Sub

**// Не возвращающий ничего метод C#.**

public void MyMethod() {

**// Некоторый программный код…**

}

Вы уже видели, что эти небольшие отличия в синтаксисе несущественны с точки зрения среды выполнения .NET, поскольку соответствующие компиляторы (в данном случае это vbc.exe и csc.exe) генерируют аналогичные множества CIL-инструкций. Но языки могут также отличаться по функциональности. Конкретный язык .NET может, например, иметь ключевое слово или не иметь его для представления данных без знака, может поддерживать или не поддерживать типы указателя. С учетом таких вариаций возможностей необходимо иметь базовый уровень, которому соответствовали бы все языки с поддержкой .NET.

Спецификации CLS (Common Language Specification – общеязыковые спецификации)- это набор правил, которые во всех подробностях описывают минимальное и полное множество возможностей, которые должен поддерживать данный .NET-компилятор, чтобы генерировать программный код, подходящий для CLR, и в то же время быть одинаково доступным для всех языков, предназначенных для платформы .NET. Во многих отношениях CLS можно рассматривать, как *подмножество*полного набора функциональных возможностей, определенного в рамках CTS.

Спецификации CLS являются в конечном счете набором правил, которых должны придерживаться создатели компилятора, если они собираются создать продукт, который во "вселенной" .NET функционирует "незаметно" для пользователя. Каждое правило имеет простое имя (например, "Правило CLS номер 6") и описывает, какое отношение это правило имеет к тем, кто создает компиляторы, и тем, кто каким-то образом взаимодействует с ними. Квинтэссенцией CLS является могущественное *правило 1.*

• *Правило 1*. Правила CLS применяются только к тем компонентам типа, которые открыты для доступа за пределами определяющего их компоновочного блока.

С учетом этого правила вы можете сделать (правильное) заключение, что все остальные правила CLS не касаются логики, используемой для внутреннего устройства типа .NET. Единственные аспекты типа, которые должны согласовываться с CLS, – это определения членов (т.е. соглашения о выборе имен, параметры и возвращаемые типы). Логика реализации конкретного члена может использовать любые несогласованные с CLS технологии, если только это будет скрыто от "внешнего мира".

Так, следующий метод Add() не является согласованным с правилами CLS, поскольку для параметров и возвращаемых значений используются данные без знака, которые в CLS не указаны.

public class Calc {

 // **Эти открытые данные без знака не согласуются с CLS!**

 public **ulong**Add(**ulong**x, **ulong**у) { return x + у; }

}

Но если вы используете данные без знака только внутри типа, как указано ниже

public class Calc {

 public int Add(int x, int y) {

**// Здесь переменная ulong используется только внутри типа,**

**// поэтому правила CLS не нарушается.**

 ulong temp;

  ...

  return x + у;

 }

}

то правила CLS остаются выполненными, и вы можете быть уверены, что теперь любой язык .NET сможет вызвать метод Add().

Конечно, кроме правила 1 в CLS определено много других правил. Например, в CLS описывается, как язык должен представлять строки текста, перечни, статические члены и т.д. К счастью, совсем не обязательно запоминать все эти правила, чтобы стать искусным разработчиком .NET. Снова повторим, что глубокое и полное понимание спецификаций CTS и CLS необходимо только создателям соответствующих инструментов разработки и компиляторов.

**Гарантия CLS-совместимости**

Как вы узнаете из текста этой книги, в C# имеется ряд программных конструкций, которые яе являются CLS-совместимыми. Однако хорошим известием является то, что вы можете заставить компилятор C# выполнять проверку вашего программного кода на соответствие CLS, используя дли этого один атрибут .NET.

**// Указание компилятору C# выполнить проверку на соответствие CLS.**

[assembly: System.CLSCompliant(true)]

В главе 12 будут рассмотрены тонкости программирования на основе использования атрибутов, Пока что важно просто понять, что атрибут [CLSCompliant] дает компилятору C# указание проверять каждую строку программного кода на соответствие правилам CLS. Если обнаружится нарушение правил CLS, вы получите сообщение об ошибке компиляции и описание некорректного программного кода.

**Общеязыковая среда выполнения**

В дополнение к спецификациям CTS и CLS, последней на данный момент аббревиатурой, которую мы рассмотрим, будет аббревиатура CLR (Common Language Runtime – общеязыковая среда выполнения). Для программирования термин *среда, выполнения*можно понимать, как набор внешних сервисов, необходимых для выполнения данной скомпилированной единицы программного кода. Например» когда разработчик при создании нового приложения использует библиотеку базовых классов Microsoft (MFC), он знает, что для выполнения его программы потребуется соответствующий выполняемый модуль библиотеки MFC (т.е. mfc42.dll). Другие популярные языки также предлагают соответствующие выполняемые модули. Программисты VB6 привязаны к одному или двум выполняемым модулям (например, msvbvm60.dll). Разработчики Java привязаны к виртуальной машине Java (JVM) и т.д.

Платформа .NET предлагает совсем другой принцип организации среды выполнения. Разница между средой выполнения .NET и средой выполнения, о которой говорилось выше, заключается в том, что среда выполнения .NET обеспечивает единый и вполне определенный "слой" выполнения, общий для *всех*языков и платформ, совместимых с .NET.

Основной механизм CLR физически заключается в библиотеке, называемой mscoree.dll (известной также под названием Common Object Runtime Execution Engine – общий объектный модуль механизма выполнения). Когда на компоновочный блок ссылаются для использования, mscoree.dll автоматически загружается и, в свою очередь, загружает в память требуемый компоновочный блок. Механизм выполнения отвечает за целый ряд задач. Прежде всего, и это самое главное, за выяснение расположения компоновочного блока и нахождение запрашиваемого типа в бинарном объекте с помощью чтения содержащихся там метаданных. Затем среда CLR размещает тип в памяти, преобразует CIL-код в соответствующие платформе инструкции, выполняет все необходимые проверки безопасности, а затем выполняет полученный программный код.

Вдобавок к загрузке созданных вами компоновочных блоков и созданию пользовательских типов, CLR также, если это необходимо, взаимодействует с типами, содержащимися в библиотеках базовых классов .NET. Хотя вся библиотека базовых классов разбита на целый ряд отдельных компоновочных блоков, "ключевым" компоновочным блоком является mscorlib.dll. Файл mscorlib.dll содержит множество базовых типов, которые объединяют в себе решения широкого спектра общих задач программирования, а также базовые типы данных, используемые всеми языками .NET. При построении .NET-приложений вы автоматически получаете доступ к этому специальному компоновочному блоку.

На рис. 1.3 показана система связей, возникающих между вашим исходным кодом (использующим типы из библиотеки базовых классов), .NET-компилятором и механизмом выполнения .NET.

**Различия между компоновочными блоками, пространствами имен и типами**

Каждый из нас понимает важность библиотек программного кода. Цель библиотек, таких как MFC, J2EE или ATL, – дать разработчику готовый набор блоков уже существующего программного кода, опираясь на которые можно строить новые приложения. Но язык C# не предлагает библиотек с программным кодом для конкретного языка. Разработчики C# могут использовать .NET-библиотеки, нейтральные в языковом отношении. Для того чтобы все типы в библиотеках базовых классов были правильно организованы, платформа .NET предлагает использовать понятие пространства имен.

Упрощенно говоря, пространство имен является группой связанных типов, содержащихся в компоновочном блоке. Например, пространство имен System.IO содержит типы, связанные с операциями ввода-вывода, пространство имен System.Data определяет основные типы для работы с базами данных и т.д. Важно понимать, что один компоновочный блок (такой как, например, mscorlib.dll) может содержать любое число пространств имен, каждое из которых может, в свою очередь, содержать любое число типов.

Чтобы ситуация стала более ясной, взгляните на рис. 1.4, на котором показан снимок окна Object Browser из Visual Studio 2005. Этот инструмент позволяет видеть компоновочные блоки, на которые ссылается текущий проект, пространства имен, содержащиеся в компоновочных блоках, типы, существующие в пределах данного пространства имен, и члены каждого типа.

Обратите внимание на то, что mscorlib.dll содержит очень много самых разных пространств имея, и в каждом из этих пространств имен содержатся свои семантически связанные типы.



Рис. 1.3. Модуль mscoree.dll в действии

Основным отличием этого подхода от таких зависящих от конкретного языка библиотек, как MFC, является то, что в результате все языки, поддерживаемые в среде выполнения .NET используют *одни и те же*пространства имен и *одни и те же* типы.

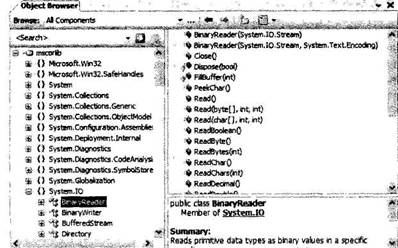


Рис.1.4. Один компоновочный блок может содержать любое количество пространств имен

Для иллюстрации рассмотрим следующие три программы, представляющие вариации вездесущего примера "Hello World" соответственно на C#, VB .NET и Managed Extensions for C++.

**// Hello world на языке C#**

using System;

public class MyApp {

 static void Main() {

  Console.WriteLine("Привет из C#");

 }

}

' **Hello world на языке VB .NET**

Imports System

Public Module MyApp

 Sub Main()

  Console.WriteLine("Привет из VB .NET")

 End Sub

End Module

// **Hello world на языке Managed Extensions for C++**

#include "stdafx.h"

using namespace System;

int main(array‹System::String^› ^args) {

 Console::WriteLine("Привет из managed C++");

 return 0;

}

Заметим, что здесь в любом из языков используется класс Console, определенный в пространстве имен System. Если отбросить незначительные синтаксические вариации, то эти три приложения выглядят очень похожими, как по форме, так и по логике.

Очевидно, вашей главной целью, как разработчика .NET. является получение исчерпывающей информации обо всем разнообразии типов, определенных в рамках (многочисленных) пространств имен .NET. Главным из пространств имен, о которых следует знать, является System. Это пространство имен предлагает базовый набор типов, которые вы, как разработчик .NET. будете использовать снова и снова. Фактически вы не сможете построить ни одного реально работающего C#-приложеиия, не сославшись, как минимум, на пространство имен System. В табл. 3.4 предлагаются краткие описания некоторых (но, конечно же, не всех) пространств имен .NET.

**Таблица 1.4.**Пространства имен .NET

|  |  |
| --- | --- |
| **Пространства имен .NET** | **Описание** |
| System | В рамках System вы найдете множество полезных типов, связанных с внутренними данными, математическими вычислениями, переменными окружения, генерированием случайных чисел и сбором мусора, а также с обработкой типичных исключительных ситуаций и атрибутов |
| System.Collections System.ColIections.Generiс | Эти пространства имен определяют ряд контейнерных объектов (ArrayList, Queue и т.д.), а также базовых типов и интерфейсов, которые позволяют строить пользовательские коллекции. В .NET 2.0 типы коллекций обладают дополнительными общими возможностями |
| System.Data System.Data.Odbc System.Data.OracleClient System.Data.OleDb System.Data.SqlClient | Эти пространства имен используются для взаимодействия с базами данных на основе ADO.NET |
| System.Diagnostics | Здесь вы найдете множество типов, которые могут использоваться для программной отладки и трассировки исходного кода |
| System.Drawing System.Drawing.Drawing2D System.Drawing.Printing | Здесь вы найдете множество типов для работы с графическими примитивами, такими как растровые изображения, шрифты и пиктограммы, а также для вывода на печать |
| System.IO System.IO.Compression System.IO.Ports | Эти пространства имен включают средства файлового ввода-вывода, буферизации и т.д. В .NET 2.0 пространства имен IO предлагают поддержку сжатия и работы с портами |
| System.Net | Это пространство имен (как и другие родственные пространства имен) содержит типы, связанные с сетевым программированием (запросы/ответы, сокеты, конечные точки соединений и т.д.) |
| System.Reflection System.Reflection.Emit | Эти пространства имен определяют типы, связанные с обнаружением типов в среде выполнения и динамическим созданием типов |
| System.Runtime.InteropServices | Это пространство имен обеспечивает средства взаимодействия типов .NET с "неуправляемым программным кодом" (это, например, DLL на базе C и COM-серверы) |
| System.Runtime.Remoting | Это пространство имен (среди прочих) определяет типы, используемые для построения решений на основе возможностей слоя удаленного доступа .NET |
| System.Security | Безопасность – это неотъемлемый аспект платформы .NET. В пространствах имен, объединенных идеей безопасности, вы найдете множество типов, связанных с разрешением доступа, криптографической защитой и т.д. |
| System.Threading | Это пространство имен определяет типы, используемые при построении многопоточных приложений |
| System.Web | Ряд пространств имен, специально предназначенных для разработки Web-приложений .NET, включая Web-сервисы ASP.NET и XML |
| System.Windows.Forms | Это пространство имен содержит типы, которые упрощают процесс создания традиционных GUI-приложений (приложений с графическим интерфейсом) для настольных систем |
| System.Xml | Связанные с XML пространства имен, содержащие множество типов, используемых для взаимодействия с XML-данными |

**Программный доступ к пространствам имен**

Не помешает повторить снова, что пространство имен является ничем иным, как подходящим для человека способом логически понять и организовать связанные типы. Снова рассмотрим пространство имен System. Можно считать, что System.Console представляет класс, названный *Console*и содержащийся в пространстве имен System. Но для среды выполнения .NET это не так. Механизм среды выполнения видит только отдельную единицу, названную System.Console.

В C# ключевое слово using упрощает ссылки на типы, определенные в данном пространстве имен. Вот как оно используется. Допустим, вы строите традиционное приложение для настольной системы. В главном окне должна отображаться некоторая диаграмма, основанная на информации, полученной от внутренней базы данных, и логотип компании. Конечно, для изучения типов в пространствах имен требуется время, но вот несколько очевидных "кандидатов", подходящих для ссылок в описанной выше программе.

**// Пространства имен, необходимые для данного приложения.**

using System; // Общие типы базовых классов,

using System.Drawing; // визуализация графики,

using System.Windows.Forms; // GDI-элементы,

using System.Data; // общий доступ к данным,

using System.Data.SqlClient; // доступ к данным MS SQL Server.

Выбрав некоторое число пространств имен (и указав ссылки на компоновочные блоки, которые их определяют), вы получаете возможность создавать экземпляры соответствующих типов. Например, если вы хотите создать экземпляр класса Bitmap (из пространства имен System.Drawing), вы можете использовать следующий программный коя,

**// Явный список пространств имен, используемых в данном файле.**

using System;

using System.Drawing;

class MyApp {

 public void DisplayLogo() {

**// Создание изображения 20x20.**

  Bitmap companyLogo = new Bitmap(20, 20);

  ...

 }

}

Поскольку ваше приложение ссылается на System.Drawing, компилятор сможет выяснить, что класс Bitmap является членом указанного пространства имен. Если бы вы не указали здесь пространство имен System.Drawing, то получили бы ошибку компиляций. Но для определения переменных можно также использовать полностью *определённое,*или *абсолютное имя.*

**// Здесь System.Drawing не указывается!**

Using System;

class MyApp {

 public void DisplayLogo() {

**// Использование абсолютного имени**

  System.Drawing.Bitmap companyLogo = new System.Drawing.Bitmap(20, 20);

  …

 }

}

Использование абсолютных имен для определения типа, конечно, упрощает понимание программного кода, но я думаю, и вы не станете возражать, что ключевое слово C# using уменьшает объем необходимого ввода. В этом тексте мы будем избегать использования абсолютных имен (за исключением тех случаев, когда из-за этого возникает явная неоднозначность), отдавая предпочтение использованию ключевого слова using.

При этом следует понимать, что использование using является лишь альтернативой использования абсолютных имен, поскольку в программном коде CIL всегда используются абсолютные имена, так что оба указанных подхода дают *совершенно одинаковые* результаты в CIL и с точки зрения производительности, и с точки зрения размера компоновочного блока.

**Ссылки на внешние компоновочные блоки**

Вдобавок к указанию пространства имен с помощью ключевого слова C# using, необходимо указать компилятору C# имя компоновочного блока, содержащего реальное CIL-определение соответствующего типа. Выше уже упоминалось, что многие базовые пространства имен .NET содержатся в mscorlib.dll. Однако тип System.Drawing.Bitmap содержится в другом компоновочном блоке с именем System.Drawing.dll. Большинство компоновочных блоков .NET Framework размещается в специальном каталоге, называемом GAC (Global Assembly Cache - глобальный кэш компоновочных блоков). На машинах Windows это может быть каталог %windir%\Assembly, как показано на рис. 1.5.

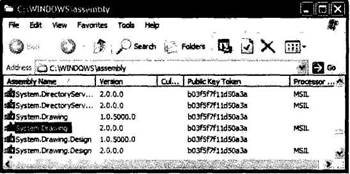


Рис. 1.5. Библиотеки базовых классов в GAC

В зависимости от используемого инструмента разработки приложений .NET может быть несколько разных способов информирования компилятора о том, какие компоновочные блоки следует включить в цикл компиляции. Подробно это будет обсуждаться в следующей главе.

**Использование ildasm.exe**

Если вас пугает перспектива освоения всех пространств имен платформы .NET, вспомните о том, что уникальность любого пространства имен заключается в том, что оно содержит типы, некоторым образом *семантически связанные*между собой. Поэтому, например, если вам не нужен интерфейс пользователя для простого консольного приложения, то смело можете забыть (среди прочих) о пространствах имен System.Windows.Forms и System.Web. Если вы строите приложение для работы с изображениями, то вам вряд ли понадобятся пространства имен для работы с базами данных. К тому же, как в случае любой новой библиотеки готового программного кода, вы можете учиться по ходу дела.

Утилита ildasm.exe (Intermediate Language Disassembler utility – утилита дизассемблера промежуточного языка) позволяет загрузить любой компоновочный блок .NET и исследовать его содержимое, включая соответствующий манифест, программный код CIL и метаданные типов.

По умолчанию файл ildasm.exe должен быть установлен в папку C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin (если вы не можете найти ildasm.exe в указанном месте, просто выполните поиск файла по ключу "ildasm.exe" на своей машине).

Обнаружив и запустив этот файл, а открывшемся окне выберите команду меню File→Open и перейдите к компоновочному блоку, который вы хотите исследовать. С целью иллюстрации здесь (рис. 1.6) показав компоновочный блок CSharpCalculator.exe, о котором уже шла речь выше. Утилита ildasm.exe представляет структуру компоновочного блока, используя всем знакомый формат дерева просмотра.

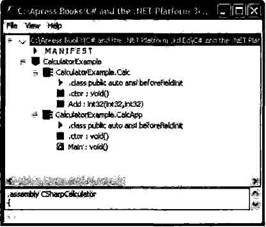


Рис. 1.6. Ваш новый лучший друг ildasm.exe

**Просмотр CIL-кода**

В дополнение к тому, что вы можете видеть пространства имен, типы и их члены в компоновочном блоке. Ildasm.exe дозволяет также просмотреть CIL-инструкции любого члена. Например, если выбрать двойным щелчком метод Main() класса CalcApp, то появится отдельное окно, в котором будет отображаться соответствующий CIL-код (рис. 1.7).

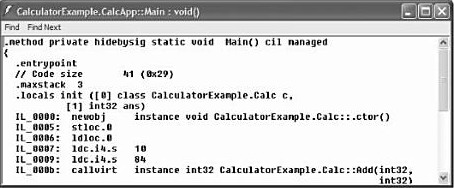


Рис. 1.7. Просмотр CIL-кода

**Просмотр метаданных типов**

Если вы захотите просмотреть метаданные типов для загруженного в настоящий момент компоновочного блока, просто нажмите ‹Ctrl+M›. На рис. 1.8 показаны метаданные для метода Calc.Add().

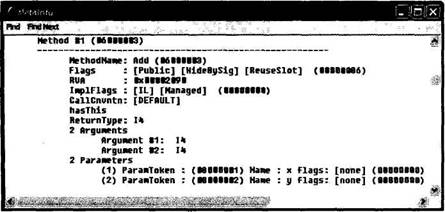


Рис. 1.8. Просмотр метаданных типов с помощью ildasm.exe

**Просмотр метаданных компоновочных блоков**

Наконец, если вы захотите просмотреть содержимое манифеста компоновочного блока, то вам нужно двойным щелчком открыть пиктограмму MANIFEST (рис. 1.9).

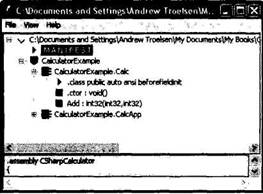


Рис. 1.9. Двойной щелчок на этой строке покажет манифест компоновочного блока

Не сомневайтесь в том, что ildasm.exe имеет не только те опции, о которых говорилось выше. Дополнительные возможности этого инструмента будут обсуждаться и дальше. По мере чтения материала книги я рекомендую вам просматривать ваши компоновочные блоки с помощью ildasm.exe, чтобы вы видели, как ваш программный код C# транслируется в независимый от платформы программный код CIL. Конечно, чтобы быть суперзвездой C#, совсем не обязательно быть экспертом по программному коду CIL, но понимание синтаксиса CIL только укрепит ваши "мускулы программирования".

**Инсталляция среды выполнения .NET**

Для вас не должно быть сюрпризом то, что компоновочные блоки .NET могут выполняться только на машине, на которой установлен каркас .NET Framework. Для вас, как для разработчика .NET-приложений, это не должно быть проблемой, поскольку ваша машина будет должным образом сконфигурирована уже в процессе установки свободно доступного пакета .NET Framework 2.0 SDK (или любой коммерческий среды разработки .NET-приложений, например Visual Studio 2005).

Однако, если вы развернете компоновочный блок на компьютере, который не имеет установленной системы .NET, этот компоновочный блок выполняться не сможет. По Этой причине Microsoft предлагает установочный пакет dotnetfx.exe, который может быть бесплатно получен и установлен вместе с вашим программным обеспечением. Эта установочная программа включена в .NET Framework 2.0 SDK, а также доступна для бесплатной загрузки с узла Microsoft.

После установки dotnetfx.exe ваша машина будет содержать библиотеки базовьх классов .NET, файлы среды выполнения .NET (mscoree.dll) и дополнительно инфраструктуру .NET (например, GAC).

**Замечание.**При построении Web-приложений .NET не предполагается, что на машине конечного пользователя будет установлен каркас .NET Framework, поскольку браузер конечного пользователи просто получает общий HTML-код и, возможно, JavaScript-код клиента.

**Платформенная независимость .NET**

В завершение этой ставы позвольте мне сказать несколько слов по поводу независимости платформы .NET. Неожиданностью для многих разработчиков является то, что компоновочные блоки .NET могут разрабатываться и выполняться в операционных системах, отличных от операционных систем Microsoft (Mac OS X, многочисленные вариации Linux, в частности BeOS и FreeBSD и др.). Чтобы поднять, почему это возможно, мы с вами должны рассмотреть еще одну аббревиатуру, используемую во "вселенной" .NET: это аббревиатура CLI (Common Language infrastructure – общеязыковая инфраструктура).

Когда Microsoft выпустила язык программирования C# и платформу .NET, она выпустила также множество формальных документов, которые описали синтаксис и семантику языков C# и CIL, формат компоновочного блока .NET, базовые пространства имен и работу гипотетического механизма среды выполнения .NET (известного также под названием VES, или Virtual Execution System – виртуальная система выполнения). Еще лучше то, что все эти документы представлены в организации Ecma International (http://www.ecma-internatiоnal.org) для утверждения их в качестве официальных международных стандартов. Интересующими нас спецификациями являются:

• ECMA-334: спецификации языка C#;

• ECMA-335: общеязыковая инфраструктура (CLI).

Важность этих документов проясняется, если заметить, что они предоставляют третьим сторонам возможность строить дистрибутивы платформы .NET для любого числа операционных систем и/или процессоров. Спецификации ECMA-335, наверное, более "содержательны", поэтому они разбиты на пять разделов, как показано в табл. 1.5.

Таблица 1.5. Разделы CLI

|  |  |
| --- | --- |
| **Разделы ECMA-335** | **Описание** |
| Раздел I. Архитектура | Описывает общую архитектуру CLI, включая правила CTS и CLS, а также работу механизма среды выполнения .NET |
| Раздел II. Метаданные | Описывает структуру метаданных .NET |
| Раздел III. CIL | Описывает синтаксис и семантику CIL-кода |
| Раздел IV. Библиотеки | Дает высокоуровневый обзор минимальных и полных библиотек классов, которые должны поддерживаться дистрибутивом .NET |
| Раздел V. Дополнения | Коллекция "вспомогательных" элементов, таких гак рекомендации по проектированию библиотек классов и подробности реализации компилятора CIL |

Заметим, что в разделе IV (Библиотеки) определяется минимальное множество пространств имен, которые представляют только базовые сервисы, ожидаемые от CLI-дистрибутива (коллекции, консольный ввод-вывод, файловый ввод-вывод, поточная обработка, отображение, сетевой доступ, базовые средства защиты, XML-манипуляции и т.д.). Такой CLI-дистрибутив не определяет пространства имен, упрощающих разработку Web-приложений (ASP.NET), доступ к базам данных (ADO.NET) или построение графического интерфейса пользователя (Windows Forms).

Благоприятным, однако, является то, что главные дистрибутивы .NET распространяют библиотеки CLI с эквивалентами Microsoft для ASP.NET, ADO.NET и Windows Forms, чтобы обеспечить полноценные платформы разработки производственного уровня. На сегодняшний день есть две главные реализации CLI (кроме предложений Microsoft, предназначенных только для Windows). Хотя в этой книге рассматривается создание .NET-приложений с помощью .NET-дистрибутива Microsoft, в табл. 1.6 представлена также информация о проектах Mono и Portable.NET.

Как Mono, так и Portable.NET обеспечивают ECMA-совместимость компилятора C# и механизма выполнения .NET, примеры программного кода, документацию, а также многочисленные инструменты разработки приложений, которые по функциональности эквивалентны инструментам, входящим в состав .NET Framework 2.0 SDK от Microsoft. К тому же Mono и Portable.NET поставляются с компиляторами VB .NET, Java и C.

**Замечание.**Если вы хотите узнать больше о Mono или Portable.NET, прочитайте книгу М. J. Easton и Jason King, *Cross-Platform .NET Development: Using Mono, Portable.NET, and Microsoft .NET*(Apress, 2004).

Таблица 1.6. .NET-дистрибутивы с открытым исходным кодом

|  |  |
| --- | --- |
| **Дистрибутив** | **Описание** |
| [http://www.mono-project.com](http://www.mono-project.com/) | Проект Mono является дистрибутивом CLI с открытым исходным кодом, предназначенным для различных вариантов Linux (например, SuSE, Fedora и т.д.), а также дня Win32 и Mac OS X |
| [http://www.dotgnu.org](http://www.dotgnu.org/) | Portable.NET – это другой дистрибутив CLI с открытым исходным кодом, предназначенный для множества операционных систем. Portable.NET нацелен на то, чтобы обслуживать как можно больше операционных систем (Win32, AIX, BeOS, Mac OS X, Solaris, все главные варианты Linux и т.д.) |

**Резюме**

Целью этой главы было описание базовых концепций, необходимых для освоения остального материала этой книги. Сначала были рассмотрены ограничения и сложности технологий, появившихся до .NET, а затем был предложен обзор того, как .NET и C# пытаются упростить существующее положение вещей.

В сущности .NET можно свести к механизму среды выполнений (mscoree.dll) и библиотеке базовых классов (mscorlib.dll и сопутствующие файлы). Общеязыковая среда выполнения (CLR) способна принять любой бинарный .NET-объект (называемый компоновочным блоком), если только этот бинарный объект подчиняется правилам управляемого программного кода. Как вы убедились, компоновочные блоки содержат CIL-инcтрукций (в дополнение к метаданным типов и манифесту компоновочного блока), которые с помощью JIT-компилятора компилируются в специфические инструкции платформы. Кроме того., была выяснена роль общеязыковых спецификаций (CLS) и общей системы типов (CTS)*.*

Затем была рассмотрена, утилита ildasm.exe, а также то, как с помощью dotnetfx.exe настроить машину для использования .NET-приложений. В заключение было сказано несколько слов о независимой от платформ природе C# и .NET.

**ГЛАВА 2. Технология создания приложений на языке C#**

Как разработчик программ на языке C#, вы имеете возможность выбрать любой из множества доступных инструментов разработки .NET-приложений. Целью этой главы является обзор самых разных инструментов разработки .NET, включая, конечно же Visual Studio 2005. Однако начнется глава с рассмотрения компилятора командной строки C#, csc.exe, для работы с которым будет достаточно самого простого текстового редактора., например программы Блокнот (noteepad.exe). Кроме того, мы с вами выясним, как выполнить отладку компоновочных блоков .NET с командной строки с помощью cordbg.exe. Освоив компиляцию и отладку компоновочных блоков без графического интерфейса, мы затем выясним, как можно редактировать и компилировать файлы исходного кода C# c помощью приложения TextPad.

Конечно, с текстом этой книги можно работать, используя только csc.exe и Блокнот/TextPad, но я думаю, вы заинтересованы в освоении более широких возможностей, предлагаемых в рамках современных IDE (Integrated Development Environment – интегрированная среда разработки). Поэтому мы рассмотрим также SharpDevelop – интегрированную среду разработки с открытым исходным текстом. По функциональности она конкурирует со многими коммерческими средствами разработки .NET, обладая тем дополнительным преимуществом, что она бесплатна, А после краткого обсуждения возможностей Visual C# 2005 Express мы приступим к рассмотрению Visual Studio 2005. Закончится глава небольшим обзором целого ряда дополнительных инструментов разработки .NET (многие из которых имеют открытый исходный код) и рекомендациями по поводу того, как эти инструменты получить.

**Установка .NET Framework 2.0 SDK**

Прежде чем начать строить .NET-приложения, используя язык программирования C# и каркас разработки приложений .NET Framework, сначала нужно установить свободно доступный пакет .NET Framework 2.0 SDK (Software Development Kit – комплект средств разработки программного обеспечения). Следует знать о том, что .NET Framework 2.0 SDK автоматически устанавливается при установке Visual Studio 2005 или Visual C# 2005 Express, поэтому если вы планируете установить одну из указанных систем, то загружать и отдельно устанавливать пакет программ .NET Framework 2.0 SDK нет необходимости.

Если у вас нет Visual Studio 2005 или Visual C# 2005 Express, то откройте страницу http://msdn.microsoft.com/netframework и выполните поиск по ключу ".NET Framework 2.0 SDK". Перейдя на соответствующую страницу, загрузите setup.exe и сохраните этот файл в подходящей папке на своем жестком диске. Затем двойным щелчком запустите этот выполняемый файл, чтобы установить соответствующее программное обеспечение.

После завершения установки на вашей машине появится не только необходимая инфраструктура .NET, но и множество инструментов разработки с очень хорошей справочной системой, примерами программного кода и обучающими программами, а также различные официальные документы с описанием системы.

По умолчанию пакет .NET Framework 2.0 SDK устанавливается в каталог C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0. Там вы найдете файл StartHere.htm. который (в полном соответствии с его названием) может служить в качестве отправной точки для доступа ко всем разделам документации. В табл. 2.1 предлагаются описания некоторых подкаталогов из корневого каталога инсталляции.

**Таблица 2.1.**Подкаталоги корневого каталога установки .NET Framework 2.0 SDK

|  |  |
| --- | --- |
| **Подкаталог** | **Описание** |
| \Bin | Содержит большинство инструментов разработки .NET-приложений. В файле StartTools.htm предлагаются описания всех утилит |
| \Bootstrapper | Почти все содержимое этого каталога можно игнорировать, но следует знать, что именно здесь, в подкаталоге \Packages\DotNetFx, находится dotnetfx.exe (см. главу 1) |
| \CompactFramework | Содержит программу установки .NET Compact Framework 2.0 |
| \Samples | Содержит программу установки набора примеров .NET Framework 2.0 SDK. О том, как установить примеры, говорится в StartSamples.htm |

В дополнение к файлам, установленным в каталог C:\Program Files\ Microsoft Visual Studio 8\ SDK\v2.0, программа установки создает подкаталог Microsoft.NET\Framework в каталоге Windows. Там вы обнаружите отдельные подкаталоги для каждой версии .NET Framework, установленной на вашей машине. Внутри подкаталога, соответствующего конкретной версии, содержатся компиляторы командной строки для каждого языка, предлагаемого в рамках Microsoft .NET Framework (это CIL, C#, Visual Basic .NET, J# и JScript .NET), а также дополнительные утилиты командной строки и различные компоновочные блоки .NET.

**Компилятор командной строки для C# (csc.exe)**

Для компиляции исходного кода C# есть целый ряд возможностей. Не касаясь Visual Studio 2005 (и различных IDE сторонних производителей), здесь можно отметить компилятор командной строки для C#, csc.exe (где csc означает аббревиатуру для *C-Sharp Compiler*- компилятор C#), с помощью .NET которого можно создавать компоновочные блоки .NET. Указанный файл входит в комплект поставки .NET Framework 2.0 SDK. Вы, конечно же, не захотите создавать большие приложения с помощью компилятора командной строки, но знать, как компилировать \*.cs-файлы вручную, вcе же важно. Можно указать несколько причин, по которым вы должны иметь представление о соответствующем процессе.

• Самым очевидным является то, что вы можете просто не иметь Visual Studio 2005.

• В ваших планах может быть использование автоматизированных средств разработки, таких как MSBuild или NAnt.

• Вы можете стремиться к расширению своего понимания C#. При использовании графических средств разработки приложений вы все равно даете инструкции csc.exe о том как обрабатывать исходные файлы C#. С этой точки зрения весьма полезно знать, что происходит "за кулисами".

Полезным "побочным эффектом" работы с csc.exe является то, что вам будет проще использовать другие инструменты командной строки, входящие в комплект поставки .NET Framework 2.0 SDK. В процессе изучения материала этой книги вы увидите, что многие очень важные утилиты оказываются доступны только из командной строки.

**Настройка компилятора командной строки для C#**

Чтобы использовать компилятор командной строки для C#, нужно, чтобы ваша система могла найти файл csc.exe. Если машина сконфигурирована неправильно, то при компиляции файлов C# вам придется указать полный путь к файлу csc.exe.

Чтобы система могла компилировать файлы \*.cs из любого каталога, выполните следующие шаги (они соответствуют установке в Windows XP; в Windows NT/2000 эти шаги, будут аналогичными).

1. Щелкните на пиктограмме Мой Компьютер и выберите пункт Свойства из раскрывшегося контекстного меню.

2. Выберите вкладку Дополнительно и щелкните на кнопке Переменные среды.

3. Двойным щелчком на имени переменной Path в окне Системные переменные откройте окно ее изменения.

4. Добавьте в конец текущего значения Path следующую строку (не забудьте о том, что значения в списке переменной Path разделяются точкой с запятой)

C:\Windows\Microsoft.NET\Framework\v2.0.50215

Ваша строка должна соответствовать версии и месту размещения .NET Framework 2.0 SDK в вашей системе (проверьие павильность указанной вами строки в окне программы Проводник). Обновив переменную Path, можно проверить результат. Для этого следует закрыть все командные окна (чтобы изменения были приняты системой), а затем, открыв новое командное окно, ввести в нём

csc /?

Если все было сделано правильно, вы должны увидеть список опций настройки, поддерживаемых компилятором C#.

**Замечание.**В списке аргументов командной строки для любого средства разработки .NET в качестве разделителя можно использовать – или / (например, csc -? или csc /?).

**Дополнительные средства командной строки .NET**

До начала использования csc.exe добавьте в системную переменную Path следующее значение (снова не забудьте проверить правильность указанного пути).

C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin

Напомним, что этот каталог содержит инструменты командной строки, которые используются в процессе разработки .NET-приложений. После создания двух указанных путей поиска у вас должна появиться возможность выполнять любую утилиту .NET из любого командного окна. Чтобы проверить правильность внесенных изменений, закройте все открытые командные окна, затем откройте новое командное окно и введите в нем следующую команду, позволяющую просмотреть опций утилиты GAC, gacutil.exe.

gacutil /?

**Совет.**Теперь вы знаете, как вручную настроить свою машину, но есть и более короткий путь. Среда .NET Framework 2.0 SDK предлагает уже сконфигурированное командное окно, распознающее все утилиты командной строки .NET. Используя кнопку Пуск, выберите из меню Все Программы→Microsoft .NET Framework SDK v2.0 и активизируйте строку SDK Command Prompt (Командная строка SDK).

**Компоновка C#-приложений с помощью csc.exe**

Теперь, когда машина распознает csc.exe, с помощью компилятора командной строки C# и программы Блокнот мы построим простой одномодульный компоновочный блок, который назовем TestApp.exe. Для начала нам нужен исходный код. Откройте программу Блокнот и введите в ее окно следующее.

**// Простое приложение на языке C#.**

using System;

class TestApp {

 public static void Main() {

  Console.WriteLine("Проверка! 1, 2, 3");

 }

}

Завершив ввод, сохраните файл с именем TestApp.cs в подходящем месте на диске (например, в каталоге C:\CscExample). Теперь рассмотрим основные опции компилятора C#. Прежде всего нужно понять, как указывается имя и тип создаваемого компоновочного блока (это может быть, например, консольное приложение с именем MyShell.exe, библиотека программного кода с именем MathLib.dll, приложение Windows Forms с именем MyWinApp.exe и т.д.). Каждая из возможностей обозначается соответствующим флагом, передаваемым в сsc.exe в виде опции командной строки (табл. 2.2).

**Таблица 2.2.**Опции компилятора C#, указывающие выходные параметры

|  |  |
| --- | --- |
| **Опция** | **Описание** |
| /out | Используется для указания имени создаваемого компоновочного блока. По умолчанию имя компоновочного блока совпадает с именем исходного файла \*.cs (в случае \*.dll) или с именем типа, содержащего метод Main() программы (в случае \*.exe) |
| /target:exe | Используется для создания консольного приложения. Данный тип выходного файла подразумевается по умолчанию, поэтому эту опцию при построении консольного приложения можно опустить |
| /target:library | Используется для построения одномодульного компоновочного блока \*.dll |
| /target:module | Используется для построения модуля. Модули являются составляющими многомодульных компоновочных блоков (см. главу 11) |
| /target:winexe | Приложения Windows можно строить и с помощью флага /target:exe, но флаг /target:winexe исключает появление окна консоли в фоновом режиме |

Чтобы скомпилировать TestАрр.сs в консольное приложение с именем TestApp.exe, перейдите в каталог, содержащий файл с исходным кодом, и введите следующую строку команд (обратите внимание на то, что флаги командной строки должны быть указаны до имен входных файлов, а не после).

csc **/target:exe**TestApp.cs

Здесь не указан явно флаг /out, поэтому выполняемый файл будет назван TestApp.exе, поскольку класс определяющий точку входа программы (метод Main()), у нас называется TestApp. Следует знать о том, что почти все флаги компилятора C# имеют сокращенные версии написания. Например, можно использовать /t вместо /target (все сокращения можно увидеть с помощью ввода csс /? в командной строке).

csc **/t:exe**TestApp.cs

К тому же, поскольку флаг /t:exe определяет вывод, используемый компилятором C# по умолчанию, для компиляции TestApp.cs можно использовать следующую простую строку.

csc TestApp.cs

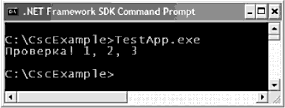


Рис. 2.1. Приложение TestApp в действии

**Ссылки на внешний компоновочный блок**

Теперь выясним, как компилировать приложение, использующее типы, определенные в отдельном компоновочном блоке .NET. Здесь, чтобы было ясно, почему при построении указанного выше приложения компилятор C# понял ссылку на тип System.Console, следует вспомнить о том (см. главу 1), что ссылка на mscorlib.dll при компиляции предполагается *автоматически.*Если же по какой-то особой причине эту ссылку необходимо отключить, следует использовать флаг /nostdlib.

Чтобы иллюстрировать механизм ссылки на внешние компоновочные блоки, мы модифицируем приложение TestApp так, чтобы оно отображало окно сообщения Windows Forms. Откройте файл TestApp.cs и измените его следующим образом.

using System;

**// Добавьте это:**

using System.Windows.Forms;

class TestApp {

 public static void Main() {

  Console.WriteLine("Проверка! 1, 2, 3");

  // **Добавьте это:**

  MessageBox.Show("Привет…");

 }

}

Здесь с помощью ключевого слова using C# (см. главу 1) добавлена ссылка на пространство имен System.Windows.Forms. Напомним, что при явном указании пространств имен, используемых в рамках файла \*.cs, нет необходимости использовать абсолютные имена (рукам легче).

В командной строке компилятору csc.exe следует сообщить о том. в каком из компоновочных блоков содержатся "используемые" пространства имен. Так, при использовании класса MessageBox с помощью опции /reference (которую можно "сократить" до /r) следует указать компоновочный блок System.Windows.Forrns.dll.

сsc **/r:System.Windows.Forms.dll**testapp.cs

Если теперь снова выполнить наше приложение, то вдобавок к выводу на консоль вы должны увидеть окно, подобное показанному на рис. 2.2.



Рис. 2.2. Ваше первое приложение Windows Forms

**Компиляция множества файлов**

В данном варианте приложение TestApp.exe использует один файл исходного кода \*.cs. Вполне возможно, чтобы все типы .NET-приложения были представлены в одном файле \*.cs, но большинство проектов компонуется из множества файлов \*.cs, чтобы программный код был более гибким. Создайте новый класс и поместите его в отдельный файл HelloMsg.cs.

// **Класс HelloMessage**

using System;

using System.Windows.Forms;

class HelloMessage {

 public void Speak() {

  MessageBox.Show("Привет…");

 }

}

Теперь обновите исходный класс ТestApp так, чтобы в нем использовался этот новый тип, а предыдущую логику Windows Forms закомментируйте.

using System;

// **Это больше не требуется:**

**// using System.Windows.Forms;**

class TestApp {

 public static void Main() {

  Console.WriteLine("Проверка! 1, 2, 3");

  // **И это тоже:**

**// MessageBox.Show("Привет…");**

**// Использование класса HelloMessage:**

  HelloMessage h = new HelloMessage();

  h.Speak();

 }

}

Скомпилируйте эти файлы C# с помощью их явного указания в качестве входных файлов.

csc /r:System.Windows.Forms.dll **testapp.cs helloimsg.cs**

В качестве альтернативы компилятор C# позволяет использовать групповой символ (\*)*,*информирующий csc.exe о том, что следует включить в текущий проект все файлы \*.cs, содержащиеся в папке проекта:

css /r:System.Windows.Forms.dll **\*.cs**

Результат выполнения новой программы не будет отличаться от предыдущего. Единственным отличием этих двух приложений будет только то, что теперь исходный код разделен на два файла.

**Ссылки на множество внешних компоновочных блоков**

В связи с рассматриваемой темой возникает следующий вопрос: "Что делать, если при использовании csc.exe нужно сослаться на множество внешних компоновочных блоков?" Просто перечислить все компоновочные блоки, используя в качестве разделителя точку с запятой. Для рассмотренного выше примера не требовалось указывать множество внешних компоновочных блоков, но вот соответствующий пример.

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

**Работа с ответными файлами csc.exe**

Очевидно, что при создании сложных C#-приложений из командной строки было бы очень неудобно набирать вручную все флаги и ссылки, необходимые для указания множества компоновочных блоков и входных файлов \*.cs. Чтобы уменьшить объемы ручного ввода, компилятор C# допускает использование *ответных файлов.*

Ответные файлы C# содержат инструкции, которые будут использоваться компилятором в процессе компиляции входного потока. По соглашению это файлы с расширением \*.rsp (сокращение от response – ответ). Предположим, что вы создали ответный файл TestApp.rsp, содержащий следующие аргументы (как видите, комментарии в данном случае обозначаются символом #).

**# Это ответный файл для TestApp.exe из главы 2.**

**# Ссылки на внешние компоновочные блоки:**

/r:System.Windows.Forms.dll

**# опции вывода и файлы для компиляции**

**# (здесь используется групповой символ):**

/target:exe /out:TestApp.exe \*.cs

Предполагая, что этот файл сохранен в каталоге с компилируемыми файлами исходного кода C#, мы можем построить наше приложение с помощью команды, показанной ниже (обратите внимание на использование символа @).

csc @TestApp.rsp

При необходимости можно указать несколько входных файлов \*.rsp (скажем, csc @FirstFile.rsp @SecondFile.rsp @ThirdFile.rsp). При таком подходе следует учитывать то, что компилятор обрабатывает командные опции в порядке их поступления. Поэтому аргументы командной строки в последнем файле \*.rsp могут "переопределить" опции предыдущих ответных файлов.

Учтите и то, что флаги, указанные явно в командной строке до ответного файла, будут "переопределены" теми флагами, которые будут указаны в соответствующем файле \*.rsp. Так, если вы введете

сsc **/out:MyCoolApp.exe** @TestApp.rsp

то именем компоновочного блока все равно будет TestApp.exe (а не MyCoolApp.exe), поскольку в ответном файле TestApp.rsp указан флаг /out:TestApp.ехe. Но если указать флаг после ответного файла, то уже флаг отменит опции ответного файла. Так, в результате выполнения следующей команды компоновочный блок получит имя MyCoolApp.exe.

csc @TestApp.rsp **/out:MyCoolApp.exe**

Замечание. Флаг /reference является кумулятивным. Независимо от того, где вы укажете внешние компоновочные блоки (до, после или внутри ответного файла), результатом будет объединение всех ссылок.

**Ответный файл, используемый по умолчанию (csc.rsp)**

В отношении ответных файлов следует знать то, что компилятор C# имеет ответный файл, используемый по умолчанию. Это файл csc.rsp, размещенный в том же каталоге, что и csc.exe (соответствующим каталогом может быть, например, C:\Windows\Microsoft.NET\Framework\v2.0.50215). Если открыть файл csc.rsp с помощью программы Блокнот, вы увидите, что в нем c помощью флага /r: уже указан целый набор компоновочных блоков .NET.

При компоновке C#-программы с помощью csc.ехe ссылка на этот фaйл выполняется автоматически, даже когда вы указываете свой файл \*.rsp. С учетом ответного файла, используемого по умолчанию, наше приложение TestApp.exe будет успешно скомпилировано и при использовании следующей команды (так как в csc.rsp есть сcылка на System.Windows.Forms.dll).

csc /out:TestApp.exe \*.cs

Если нужно отключить автоматическое чтение файла csc.rsp*,*следует указать опцию /noconfig.

csc @TestApp.rsр **/noconfig**

Компилятор командной строки C# имеет множество других опций, которые можно использовать для управления процессом генерирования компоновочных блоков .NET. Если вам требуется более подробная информация о функциональных возможностях csc.exe, прочитайте мою статью "Working with the C# 2.0 Command line Compiler" (Работа с компилятором командной строки C# 2.0), которую можно найти на страницах [http://msdn.microsoft.com](http://msdn.microsoft.com/).

**Отладчик командной строки (cordbg.exe)**

Прежде чем перейти к рассмотрению возможностей компоновки C#-приложе-ний с помощью TextPad, следует отметить, что .NET Framework 2.0 SDK предлагает отладчик командной строки cordbg.ехe. Этот инструмент имеет множество опций, которые позволяют выполнить отладку компоновочного блока. Чтобы увидеть список этих опций, используйте флаг /?.

cordbg /?

В табл. 2.3 показаны некоторые (но, конечно же, не все) флаги с указанием их сокращенных форм, распознаваемые отладчиком cordbg.exe в сеансе отладки.

**Таблица 2.3.**Некоторые флаги командной строки отладчика cordbg.exe

|  |  |
| --- | --- |
| **Флаг** | **Описание** |
| b[reak] | Установить или показать текущие точки останова |
| del[ete] | Удалить одну или несколько точек останова |
| ex[it] | Выход из отладчика |
| g[o] | Продолжить отладку текущего процесса до следующей точки останова |
| o[ut] | Выйти из текущей функции |
| p[rint] | Напечатать все загруженные переменные (локальные, аргументы и т.д.) |
| si | Войти в следующую строку |
| so | Перейти через следующую строку |

Большинство из вас предпочтет использовать интегрированный отладчик Visual Studio 2005, поэтому я не собираюсь комментировать все флаги cordbg.exe. Но для тех, кому это интересно, в следующем разделе предлагается краткое пошаговое описание основных этапов процесса отладки с использованием командной строки.

**Отладка с командной строки**

Перед началом отладки приложения с помощью cordbg.exe следует сгенерировать отладочные символы для текущего приложения, указав для csc.exe флаг /debug. Например, чтобы сгенерировать данные отладки для приложения TestApp.exe, введите следующую команду.

csc @testapp.rsp **/debug**

В результате генерируется новый файл, в данном случае с именем testapp.pdb. Без соответствующего файла \*.pdb использовать cordbg.exe тоже можно, но при этом в процессе отладки вы не сможете видеть исходный код C# (что, как правило, важно, если вы не хотите усложнять себе жизнь чтением программного кода CIL).

Сгенерировав файл \*.pdb, откройте сеанс отладки, указав для cordbg.exe свой компоновочный блок .NET в виде аргумента командной строки (при этом файл \*.pdb будет загружен автоматически).

cordbg.exe testapp.exe

Начнется режим отладки, и вы получите возможность применять любые допустимые флаги cordbg.exe в командной строке (cordbg) (рис. 2.3).

Чтобы выйти из режима отладки cordbg.exe, следует просто ввести exit (или, сокращенно, ех). Если вы не являетесь стойким приверженцем использования командной строки, вы предпочтете использовать возможности графического отладчика, предлагаемого интегрированной средой разработки. В любом случае для получения дополнительной информации обратитесь к разделу в документации .NET Framework 2.0 SDK, посвященному cordbg.exe.

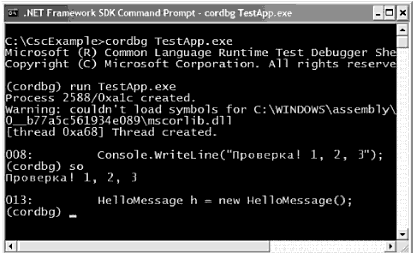


Рис. 2.3. Отладка приложения с помощью cordbg.exe

**Компоновка .NET-приложений с помощью TextPad**

Бесплатный редактор Блокнот, несомненно, подходит для создания простых программ .NET, но он не может ничего предложить для повышения производительности труда разработчика. Хорошо, когда редактор, с помощью которого создаются файлы \*.cs, поддерживает (как минимум) выделение цветом ключевых слов и блоков программного кода, а также предлагает интеграцию с компилятором C#. Как и следует ожидать, такой инструмент существует – это TextPad.

Редактор TextPad можно использовать для создания и компиляции программного кода не только на языке C#, но и многих других языках программирования. Главное преимущество этого продукта заключается в той, что он, с одной стороны, очень прост в использовании, а с другой – обеспечивает достаточно широкие возможности для упрощения процесса создания программного кода.

Чтобы получить TextPad, перейдите на страницу http://www.textpad.com и загрузите текущую версию этого редактора (во время создания нашей книги это была версия 4.7.3). Установив этот редактор, вы сразу получите полноценную версию TextPad с полным набором его возможностей, но знайте, что этот продукт не бесплатен. Пока вы не купите лицензию (ее стоимость около $30 для одного пользователя), вы будете видеть "дружеские напоминания" при каждом запуске этого приложения.

**Активизация цветовой схемы C#**

Изначально редактор TextPad не настроен на понимание ключевых слов C# и работу с сsc.exe. Чтобы настроить его соответствующим образом, нужно установить подходящее расширение. Откройте страницу http://www.textpad.com/add-ons/syna2g.html и загрузите файл csharp8.zip по ссылке C# 2005. Соответствующее расширение учитывает новые ключевые слова, введенные в C# 2005 (в отличие от файла, загружаемого по ссылке C#, в котором учитываются только возможности C# 1.1).

Развернув архив csharp8.zip, поместите копию извлеченного файла csharp8.syn в подкаталог Samples каталога инсталляции TextPad (например, в C:\Program Files\TextPad 4\Samples). Затем запустите TextPad и с помощью New Document Wizard (Мастер создания нового документа) выполните следующие действия.

1. Выберите Configured New Document Class из меню.

2. Введите имя C# 2.0 в поле редактирования Document class name (Имя документа класса).

3. Затем введите \*.cs в поле редактирования Class members (Члены класса).

4. Активизируйте подсветку синтаксиса, выберите csharp8.syn из раскрывающегося списка и закройте окно мастера.

Теперь вы можете настроить поддержку C# в TextPad, используя узел Document Classes (Классы документа), доступный из меню Configure**→**Preferences (рис. 2.4).

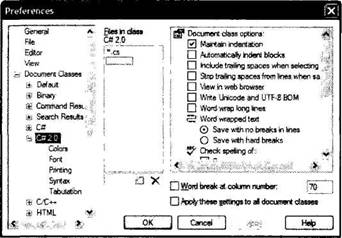


Рис. 2.4. Установка параметров редактора TextPad

**Настройка фильтра файлов \*.cs**

Следующим шагом конфигураций является создание фильтра для файлов исходного кода C#, отображаемых в диалоговых окнах Open (Открытие документа) и Save (Сохранение документа).

1. Сначала выберите Configure**→**Preferences из меню, а затем – элемент File Name Filters (Фильтры имен файлов) дерева просмотра.

2. Щелкните на кнопке New (Создать), а затем введите C# в поле Description (Описание) и \*.cs в текстовый блок Wild cards (Групповые символы).

3. Переместите свой новый фильтр в начало списка, используя для этого кнопку Move Up (Вверх), а затем щелкните на кнопке ОК.

Создайте новый файл (используйте File**→**New) и сохраните его в подходящем месте на диске (например, в папке C:\TextPadTestApp) под именем TextPadTest.cs. Затем введите тривиальное определение класса (рис. 2.5).

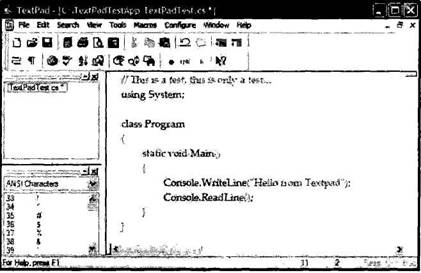


Рис. 2.5. Файл TextPadTest.cs

**Подключение csc.exe**

Последним из оcновных шагов конфигурации редактора TextPad будет связь с сsc.exe которая позволит компилировать C#-файлы. С этой целью можно, например, выбрать Tools**→**Run из меню. Вы увидите диалоговое окно, которое позволит указать имя соответствующей программы и необходимые флаги командной строки. Так, чтобы скомпилировать TextPadTest.cs в выполняемый консольный файл .NET, выполните следующие шаги.

1. Введите полный путь к файлу csc.exe в текстовое поле Command (Команда), например C:\Windows\Microsoft.NET\Framework\v2.0.502l5\csc.exe.

2. Необходимые опции командной строки введите в текстовое поле Parameters (Параметры) – например, /out:myАрр.exe \*.сs. Для упрощения процесса настройки можно указать ответный файл (например, @myInput.rsp).

3. В текстовом поле Initial folder (Исходный каталог) укажите каталог содержащий входные файлы (для нашего примера это с:\TextPadTestApp).

4. Если вы хотите, чтобы редактор TextPad захватывал вывод компилятора (а не показывал его в отдельном командном окне), установите флажок Capture Output (Захват вывода).

На риc. 2.6 показаны все необходимые для вашего примера установки компиляции.



Рис. 2.6. Установка параметров команды Run

Свою программу вы можете запустить либо с помощью двойного щелчка на имени ее выполняемого файла в программе Проводник Windows, либо с помощью выбора Tools**→**Run из меню редактора TextPad, указав myApp.exe в качестве текущей команды (рис. 2.7).

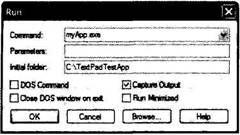


Рис. 2.7. Указание редактору TextPad запустить myApp.exe

После щелчка на кнопке ОК вы должны увидеть вывод программы ("Hello from TextPad"), отображенный в документе Command Results (Результаты команды).

**Ассоциация команд с пунктами меню**

Редактор TextPad также позволяет создавать пункты пользовательского меню, представляющие заданные команды. Для выполнения компиляции всех C#-файлов в текущем каталоге мы создадим новый пункт меню Compile C# Console (Консоль компиляции C#) в меню Tools (Сервис).

1. Сначала выберите Configured Preferences из меню, а затем – элемент Tools дерева просмотра.

2. С помощью кнопки Add (Добавить) выберите Program (Программа) и укажите полный путь к csc.exe.

3. Вместо csc.exe можно указать для меню более информативную строку, – например Compile C# Console, – щелкнув на соответствующем имени, после чего следует щелкнуть на кнопке ОК.

4. Наконец, выберите Configure**→**Preferences из меню еще раз, но на этот перейдите к элементу Compile C# Console узла Tools и укажите значение \*.cs в поле Parameters (Параметры), рис. 2.8.

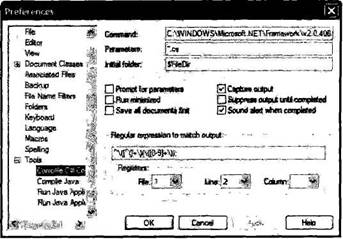


Рис. 2.8. Создание элемента меню Tools

Теперь вы получите возможность компилировать все файлы C# из текущего каталога с помощью нового пункта меню Tools.

**Использование фрагментов программного кода C#**

Перед использованием TextPad следует упомянуть еще об одном бесплатном расширений, которое вы можете установить. Перейдите на страницу http://www.textpad.com*/*add-ons/cliplibs.html и загрузите файл csharp\_l.zip с библиотекой фрагментов C#, которую предлагает Шон Гефарт (Sean Gephardt). Извлеките из архива файл csharp.tсl и поместите этот файл в подкаталог Samples. Снова запустив TextPad, вы обнаружите новую библиотеку фрагментов программного кода C Sharp Helpers, доступную из раскрывающегося списка Clip Library (Библиотека фрагментов), рис. 2.9. С помощью двойного щелчка на любом из ее элементов вы можете добавить соответствующий программный код C# в той точке активного документа, где в настоящий момент находится курсор.

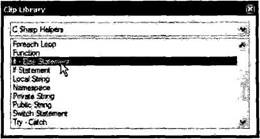


Рис. 2.9. Фрагменты программного кода C# в TextPad

Наверное, вы не станете возражать, что по сравнению с программой Блокнот и командной строкой использование редактора TextPad – шаг в правильном направлении. Однако TextPad (пока что) не предлагает возможности IntelliSense для программного кода C#, графических средств разработки, шаблонов проектов и средств работы с базами данных. Чтобы представить такие возможности, рассмотрим следующий инструмент .NET-разработкш SharpDevelop.

**Компоновка .NET-приложений с помощью SharpDevelop**

SharpDevelop является интегрированной средой разработки с открытым исходным кодом и богатыми возможностями, которые вы можете использовать для создания компоновочных блоков .NET на основе C#, VB .NET, Managed Extensions для C++ или CIL. Кроме того, что эта среда разработки совершенно бесплатна, следует отметить то, что она целиком создана на языке C#. Причем вы можете либо загрузить и скомпилировать необходимые файлы \*.cs самостоятельно, либо использовать готовую программу setup.exe, которая установит SharpDevelop на вашей машине. Оба дистрибутива можно загрузить со страниц http:// www.icsharpcode.net/OpenSource/SD/Download.

После установки SharpDevelop выбор меню File**→**New**→**Combine позволит указать вид (и язык .NET) проекта, который вы хотите создать. В терминах SharpDevelop *combine*(комбинат) обозначает отдельную коллекцию проектов – то, что в Visual Studio называется *solution,*т.е. решение. Предположим, что вы указали C#-прило-жение для Windows и назвали его MySDWinApp (рис. 2.10).

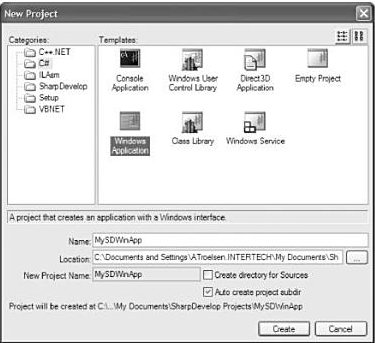


Рис. 2.10. Диалоговое окно создания проекта в SharpDevelop

**Замечание.**Приложение SharpDevelop версии 1.0 настроена на использование компилятора C# 1.1. Чтобы использовать новый возможности языка C# 2005 и пространства имён .NET Framework 2.0, выберите Project→Project options из меню и укажите новую версию компилятора на странице настроек Runtime/Compiler (Среда выполнения/компилятор).

**Возможности SharpDevelop**

Среда разработки SharpDevelop предлагает разнообразные возможности повышения производительности труда программиста, и во многих отношениях эта среда разработки столь же богата возможностями, как и Visual Studio .NET 2003 (но не настолько, как Visual Studio 2005). Вот список основных преимуществ SharpDevelop:

• поддержка компиляторов C# от Microsoft и Mono;

• возможности IntelliSense и расширения программного кода;

• наличие диалогового окна Add Reference (Добавление ссылки) для ссылок на внешние компоновочные блоки, включая компоновочные блоки, установленные в GAG (Global Assembly Cache – глобальный кэш компоновочных блоков);

• наличие инструментов визуального проектирования Windows Forms;

• различные окна (в SharpDevelop они называются scouts – разведчики) для обзора структуры проекта и его составляющих:

• интегрированная утилита браузера объектов – Assembly Scout (Разведчик компоновочных блоков);

• утилиты для работы с базами данных;

• утилита конвертирования программного кода C# в VB .NET (и наоборот):

• интеграция с NUnit (утилита тестирования .NET-модулей) и NAnt (утилита компоновки .NET):

• интеграция с документацией .NET Framework SDK.

Впечатляюще для бесплатной IDE, не так ли? В этой главе мы не собираемся обсуждать каждый из указанных пунктов подробно, но давайте рассмотрим наиболее интересные из них. Если вас интересуют подробности, то заметим, что SharpDevelop предлагает очень подробную документацию, доступную при выборе Help→Help Topics из меню.

**Окна проектов и классов**

Создав новый комбинат, вы можете использовать окно Projects для просмотра файлов, ссылок и ресурсов соответcтвующих проектов (рис. 2*.*11).



Рис. 2.11. Окно проектов

Чтобы в текущем проекте сослаться на внешний компоновочный блок, в окне Projects щелкните правой кнопкой мыши на пиктограмме References (Ссылки) и из появившегося контекстного меню выберите Add Reference (Добавить ссылку). После этого на вкладке GAC или .NET Assembly Browser (Обзор компоновочных блоков .NET) вы сможете выбрать компоновочный блок, размещенный, соответственно, в GAC или в другом месте (рис. 2.12).

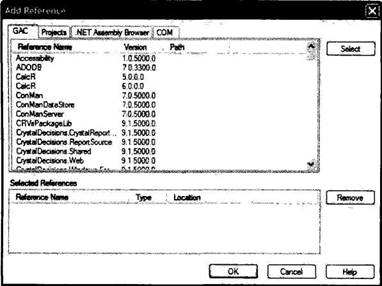


Рис. 2.12. Диалоговое окно добавления ссылок в SharpDevelop

Окно Classes обеспечивает объектно-ориентированный взгляд на комбинат, отображая пространства имен, типы и члены типов, определенные в рамках проекта (рис. 2.13).

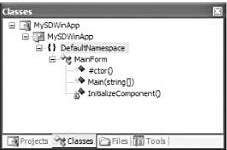


Рис. 2.13. Окно классов

Двойной щелчок на любом элементе открывает соответствующий файл, помещая курсор мыши на определение элемента.

**Обзор компоновочных блоков**

Утилита Assembly Scout [Разведчик компоновочных блоков), доступная из меню View, предлагает обзор компоновочных блоков, на которые имеются ссылки в проекте. Это средство предлагает информацию в двух панелях, Левая панель предлагает дерево просмотра, позволяющее "войти" внутрь компоновочного блока, чтобы увидеть пространства имен и соответствующие типы (рис. 2.14).

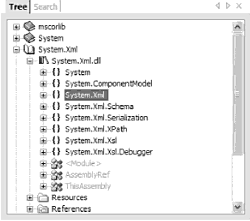


Рис. 2.14. Просмотр компоновочных блоков в окне Assembly Scout

Правая панель утилиты позволяет увидеть содержимое элемента, выбранного в левой панели. При этом можно увидеть не только основные характеристики элемента, используя для этого вкладку Info (Информация), но и соответствующий программный код CIL. Можно также сохранить определение элемента в файле XML.

**Инструменты проектирования Windows Forms**

Windows Forms является средством создания приложений, возможности которого мы рассмотрим позже. А сейчас, чтобы продолжить рассмотрение SharpDevelop, щелкните на ярлыке вкладки Design внизу окна программного кода MainForm.cs. Откроется интегрированное окно проектирования Windows Forms,

С помощью элементов из раздела Windows Forms в окне Tools можно построить графический интерфейс (GUI) для создаваемой формы. Для примера поместите один элемент типа Button (кнопка) в свою главную форму, сначала выбрав пиктограмму Button, а затем щелкнув в окне проектирования. Для изменения вида любого элемента, можно использовать окно Properties (Свойства), которое активизируется с помощью выбора View**→**Properties из меню (рис. 2.15). Выберите Button из раскрывающегося списка этого окна и укажите нужные параметры для свойств этого типа (например, BackСolor или Text).

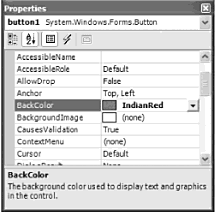


Рис. 2.15. Окно свойств

В этом же окне можно указать параметры обработки событий соответствующего элемента графического интерфейса. Для этого сначала щелкните на пиктограмме с изображением молнии (вверху окна свойств), а затем выберите из раскрывающегося списка графический элемент, с которым вы хотите работать (в данном случае это Button). Наконец, задайте правила обработки события Click (щелчок), напечатав имя метода, который должен вызываться при каждом щелчке пользователя на данной кнопке (рис. 2.16).

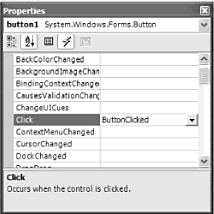


Рис. 2.16. Установка правил обработки событий в окне свойств

После нажатия клавиши ‹Enter› SharpDevelop сгенерирует программный код заглушки для вашего нового метода. Для данного примера добавите в программу обработки события следующий программный код.

void ButtonClicked(object sender, System.EventArgs e) {

***//*Поместим в заголовок формы новое сообщение.**

 this.Text = "Прекратите щелкать на моей кнопке!";

}

Теперь можно запустить программу на выполнение (выбрав Debug**→**Run из меню). Ясно, что в данном случае при щелчке на кнопке мы должны увидеть изменившийся заголовок окна формы.

Сейчас вы должны обладать достаточной информацией для того, чтобы начать использование интегрированной среды разработки SharpDevelop. Я надеюсь, что вы смогли представить себе ее основные возможности, хотя, очевидно, этот инструмент может предложить гораздо больше, чем было показано здесь.

**Компоновка .NET-приложений с помощью Visual C# 2005 Express**

Летом 2004 года Microsoft предложила совершенно новую серию IDE-продуктов, обозначенную еловом "Express" (см. http://msdn.microsoft.com/express). На сегодня выпущено шесть пакетов этого семейства.

• *Visual Web Developer 2005 Express.*"Облегченный" вариант средств разработки динамических Web-узлов и Web-сервисов XML, использующих ASP.NET 2.0.

• *Visual Basic 2005 Express.*Инструменты программирования, идеальные для программистов без большого опыта, которые хотят научиться строить .NET-приложения с помощью дружественного синтаксиса Visual Basic .NET.

• *Visual C# 2005 Express. Visual C++ 2005 Express и Visual J# 2005 Express.*Специальные инструменты разработки для учащихся и энтузиастов, предпочитающих изучать основы информатики в рамках синтаксиса соответствующего языка.

• *SQL Server 2005 Express.*Система управления базами данных начального уровня, предназначенная для любителей, энтузиастов и учащихся-разработчиков.

Замечание. Во время подготовки этой книги к печати семейство продуктов Express в виде бета-версий предлагалось совершенно бесплатно.

По большому счету продукты серии Express являются "редуцированными" версиями их аналогов из Visual Studio 2005 и предназначены главным образом для любителей .NET и учащихся. Как и в SharpDevelop, в Visual C# 2005 Express предлагаются различные средства просмотра, окно проектирования Windows Forms, диалоговое окно Add References (Добавление ссылок), возможности IntelliSense и шаблоны расширения программного кода. Кроме того, в Visual C# 2005 Express предлагается несколько (очень важных) возможностей, в настоящее время в SharpDevelop недоступных, а именно:

• интегрированный графический отладчик;

• средства упрощения доступа к Web-сервисам XML.

Ввиду того, что по виду и принципам использования Visual C# 2005 Express очень похож на Visual Studio 2005 (и в некоторой степени на SharpDevelop), здесь обсуждение указанной среды разработки не предлагается. Если вы хотите узнать об этом продукте больше, прочитайте мою статью "An Introduction to Programming Using Microsoft Visual C# 2005 Express Edition" (Введение в программирование с помощью Microsoft Visual C# 2005 Express Edition), доступную на страницах [http://msdn.microsoft.com](http://msdn.microsoft.com/).

**Компоновка .NET-приложений с помощью Visual Studio 2005**

Если вы являетесь профессиональным разработчиком программного обеспечения .NET, очень велика вероятность того, что ваш работодатель согласится купить для вас лучшую интегрированную систему разработки от Microsoft – Visual Studio 2005 (http://msdn.microsoft.com/vstudio). Этот инструмент по своим возможностям существенно превосходит все другие IDE, рассмотренные в этой главе. Конечно же, это отражается и на его цене, которая зависит от приобретаемой вами версии Visual Studio 2005. Нетрудно догадаться, что каждая версия предлагает свой уникальный набор возможностей.

В дальнейшем при изложении материала книги будет предполагаться, что вы предпочли использовать в качестве среды разработки Visual Studio 2005. С другой стороны, и вы должны это понимать, что обладание копией Visual Studio 2005 для изучения материала книги *не является обязательным.*Самой большой проблемой Из тех. с которыми вы можете столкнуться в таком случае, это обсуждение опций, которых нет в вашей среде разработки. Но программный код всех примеров данной книги должен компилироваться с помощью любого выбранного вами инструмента разработки.

**Замечание.**Загрузив исходный код примеров этой книги из раздела Downloads (загрузка) Web-узла Apress (http://www.apress.com), вы сможете открывать программный код примеров в Visual Studio 2005 с помощью двойного щелчка на соответствующем файле \*.sln. Если вы не используете Visual Studio 2005, вам придется вручную настроить свою среду разработки так, чтобы вы могли выполнить компиляцию соответствующих файлов \*.cs.

**Возможности Visual Studio 2005**

Как и следует ожидать, Visual Studio 2005 содержит все необходимые средства проектирования, средства доступа к базам данных, утилиты обзора проектов и объектов, а также интегрированную систему справки. Но, в отличие от средств разработки, которые были рассмотрены выше. Visual Studio 2005 предлагает много дополнительных возможностей. Вот список некоторых из них.

• Средства визуального проектирования/редактирования XML

• Поддержка разработки программ для мобильных устройств (например, для смартфонов и КПК)

• Поддержка разработки программ для Microsoft Office

• Возможность записи изменений исходного документа и просмотра таких изменений

• Интегрированная поддержка факторизации программного кода

• XML-библиотека расширений программного кода

• Визуальные средства построения классов и утилиты тестирования объектов

• Окно определений программного кода, которое предлагается вместо утилиты Windows Forms Class Viewer, wincv.exe, предлагавшейся в .NET версии 1.1 и более ранних версий

Честно говоря, Visual Studio 2005 предлагает так много возможностей, что для их полного описания требуется отдельная книга (по объему больше этой). Наша книга решает другие задачи. Тем не менее, я счел целесообразным потратить несколько страниц на описание самых главных усовершенствований, предложенных в Visual Studio 2005. По мере чтения материала книги вы будете получать дополнительную информацию о возможностях этой интегрированной среды разработки.

**Утилита обзора решений**

Если вы следуете указаниям этой главы, то создайте новое консольное приложение C# (с именем Vs2005Example), выбрав File→New→Project из меню. Утилита Solution Explorer (Утилита обзора решений), доступная из меню View, позволяет просматривать множество файлов и компоновочных блоков, из которых составлен текущий проект (рис. 2.17).

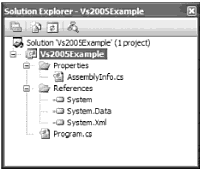


Рис. 2.17. Окно обзора решений

Обратите внимание на то, что папка References (Ссылки) в окне Solution Explorer отображает список компоновочных блоков, на которые вы ссылаетесь в настоящий момент (консольные проекты по умолчанию ссылаются на System.dll, System.Data.dll и System.Xml.dll). Если нужно сослаться на другие компоновочные блоки, щелкните правой кнопкой мыши на папке References и выберите из контекстного меню Add Reference (Добавить ссылку). В появившемся диалоговом окне вы сможете выбрать нужный вам компоновочный блок.

**Замечание.**В Visual Studio 2005 позволяется указывать ссылки на выполняемые компоновочные блоки (в отличие от Visual Studio .NET 2003, где в данном контексте можно было использовать только библиотеки программного кода *\*.*dll).

Наконец, обратите внимание на пиктограмму Properties (Свойства) в окне Solution Explorer. В результате двойного щелчка на ней появляется окно расширенного редактора конфигурации проекта (рис. 2.18).



Рис. 2.18. Окно редактирования свойств проекта

Возможности изменения параметров проекта в окне Project Properties (Свойства проекта) будут обсуждаться в книге по мере необходимости. Но даже при беглом его изучении становится ясно, что в нем можно установить параметры безопасности, изменить имя компоновочного блока, указать ресурсы командной строки и задать конфигурацию учета событий компиляции.

**Утилита обзора классов**

Следующим инструментом, который мы собираемся рассмотреть, является утилита Class View (утилита обзора классов), и ее тоже можно загрузить из меню View. Как и в случае SharpDevelop, эта утилита обеспечивает возможность обзора всех типов, входящих в текущий проект, с объектно-ориентированной точки зрения. Верхняя панель соответствующего окна отображает множество пространств имен и их типов, а нижняя панель представляет члены выбранного в настоящий момент типа (рис. 2.19).

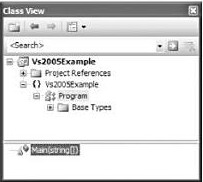


Рис. 2.19. Окно обзора классов

**Окно определений программного кода**

Если вы имеете опыт программирования в .NET 1.1, то должны знать об утилите Windows Forms Class Viewer, wincv.exe (утилита обзора классов Windows Forms). Этот инструмент позволяет задать имя .NET-типа и просмотреть его C#-определение. В версии .NET 2.0 утилиты wincv.exe уже нет, но зато есть усовершенствованная версия этого средства, интегрированная в Visual C# 2005 Express и Visual Studio 2005. Соответствующее окно Code Definition (Окно определений программного кода) можно открыть через меню View. Поместите курсор мыши на любой из типов в программном коде C#, и вы увидите определение соответствующего типа. Например, если щелкнуть на слове "string" в рамках метода Main(), будет показано определение типа класса System.String (рис. 2.20).

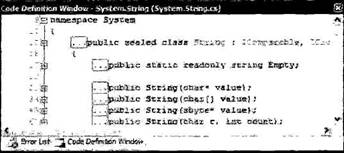


Рис. 2.20. Окно определений программного кода

**Утилита обзора объектов**

Вы должны помнить из главы 1, что в Visual Studio 2005 есть утилита для просмотра компоновочных блоков, на которые ссылается проект. Активизируйте окно Object Browser с помощью меню View→Other Windows, а затем выберите компоновочный блок, который вы желаете изучить (рис. 2.21).

**Интегрированная поддержка факторизации программного кода**

Одним из главных усовершенствований, предлагаемых в Visual Studio 2006, является встроенная поддержка факторизации программного кода. Говоря упрощенно, факторизация означает формальный "механический" процесс усовершенствования существующего базового кода. В прошлом процесс факторизации предполагал огромные объемы ручного труда. В Visual Studio 2005 значительная часть соответствующей работы выполняется автоматически. Используя меню Refaсtor (Факторизация), соответствующие комбинации клавиш, смарт-теги и/или вызовы контекстного меню с помощью щелчков мыши, вы можете придать своему программному коду совершенно новый вид. затратив на это минимум усилий. В табл. 2.4 приведены некоторые общие команды факторизации, распознаваемые в Visual Studio 2005.

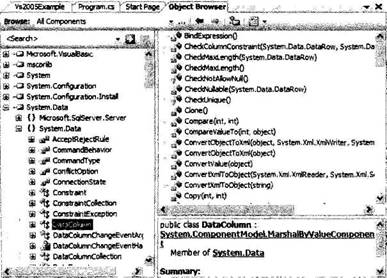


Рис. 2.21. Утилита обзора объектов в Visual Studio 2005

**Таблица 2.4.**Факторизация в Visual Studio 2005

|  |  |
| --- | --- |
| **Метод факторизации** | **Описание** |
| Extract Method (выделение метода) | Позволяет определить новый метод на основе выделенных операторов  программного кода |
| Encapsulate Field (инкапсуляция поля) | Превращает открытое поле в приватное, инкапсулированное в свойство  C# |
| Extract Interface (выделение интерфейса) | Определяет новый интерфейсный тип на основе множества  существующих членов типа |
| Reorder Parameters (перестановка параметров) | Обеспечивает изменение порядка следования аргументов |
| Remove Parameters (удаление параметров) | Удаляет данный аргумент из списка параметров |
| Rename (переименование) | Позволяет переименовать лексему программного кода (метод, поле,  локальную переменную и т.д.) |
| Promote Local Variable to Parameter | (перемещение локальной переменной в параметр) Перемещает  локальную переменную в набор параметров определяемого метода |

Чтобы проиллюстрировать возможности применения средств факторизации на практике, добавьте в метод Main() следующий программный код.

static void Main(string[] args) **{**

**// Определение консольного интерфейса (CUI)**

 Console.Title = "Мое приложение".;

 Console.ForegroundColor = ConsoleColor.Yellow;

 Console.BackgroundColor = ConsoleColor.Blue;

 Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

 Cоnsole.WriteLine("\*\*\*\*\*\*\*\* Это мое приложение! \*\*\*\*\*\*\*\*");

 Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

 Console.BackgroundColor*=*ConsoleColor.Black;

**// Ожидание нажатия клавиши для завершения работы.**

 Console.ReadLine();

}

Этот программный код вполне работоспособен, но представьте себе, что вы хотите отображать генерируемую им подсказку в разных местах вашей программы. Вместо того чтобы вводить вручную операторы определения интерфейса снова и снова, было бы идеально иметь помощника, который мог бы делать это за вас. К счастью, в данном случае вы можете применить к существующему программному коду метод факторизации Extract Method (Выделение метода). Сначала в окне редактора выберите все операторы программного кода (за исключением последнего вызова Console.ReadLine()). Затем щелкните правой кнопкой мыши и из появившегося контекстного меню Refactor выберите опцию Extract Method. В появившемся диалоговом окне укажите имя нового метода – ConfigurеCUI(). В результате вы обнаружите, что теперь метод Main() вызывает новый сгенерированный метод ConfigureCUI(), содержащий ранее выделенный программный код.

class Program {

 static void Main (string[] args) {

  ConfigureCUI();

*//*Ожидание нажатия клавиши для завершения работы.

  Cоnsole.ReadLine();

 }

 private static void ConfigureCUI() {

**// Определение консольного интерфейса (CUI)**

  Console.Title = "Мое приложение";

  Console.ForegroundColor = ConsoleColor.Yellow;

  Console.BackgroundColor = ConsoleColor.Blue;

  Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

  Cоnsole.WriteLine("\*\*\*\*\*\*\*\* Это мое приложение! \*\*\*\*\*\*\*\*");

  Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

  Console.BackgroundColor*=*ConsoleColor.Black;

 }

}

Замечание, Если вы хотите знать больше о процессе факторизации и ее поддержке в Visual Studio 2005, прочитайте мою статью "Refactoring C# Code Using Visual Studio 2005" (Факторизация программного кода C# в Visual Studio 2005), доступную на страницах [http://msdn.microsoft.com](http://msdn.microsoft.com/).

**Фрагменты программного кода и окружения**

В Visual Studio 2005 (как и в Visual C# 2005 Express) предлагаются разнообразные возможности автоматического добавления сложных блоков программного кода C# с помощью выбора вариантов меню, контекстно-зависимых щелчков кнопкой мыши и/или комбинаций клавиш. Число доступных расширений программного кода весьма впечатляюще и может быть разбито на две главные группы.

• *Фрагменты программного кода.*Это шаблоны блоков программного кода, вставляемые в том месте, где размещается курсор мыши.

• *Окружения.*Это шаблоны окружений, в которые помещаются выделенные блоки операторов в рамках соответствующего контекста.

Чтобы воспользоваться соответствующими функциональными возможностями, щелкните правой кнопкой мыши на пустой строке в пределах метода Main() и активизируйте меню Insert Snippet (Вставка фрагмента). Выбрав соответствующий пункт меню, вы увидите, что указанный программный код будет добавлен автоматически (нажмите клавишу ‹Esc›, чтобы скрыть всплывающее меню).

Если вместо этого щелкнуть правой кнопкой мыши и выбрать пункт меню Surround With (Окружить с помощью…), вы увидите другой подобный список опций. Найдите время и без спешки исследуйте встроенные шаблоны расширений программного кода, так как с их помощью можно значительно ускорить процесс разработки программ.

**Замечание.**Bce шаблоны расширений программного кода представляют собой XML-описания программного кода, генерируемые средствами IDE. В Visual Studio 2005 (и в Visual C# 2005 Express) вы можете создавать свои собственные шаблоны. Подробности этого процесса описаны в моей статье "Investigating Code Snippet Technology" (Исследование технологии применения фрагментов программного кода), которую можно найти на страницах [http://msdn.microsoft.com](http://msdn.microsoft.com/).

**Средства визуального проектирования классов**

В Visual Studio 2005 есть возможность конструировать классы визуально (в Visual C# 2005 Express такой возможности нет). Утилита Class Designer позволяет просматривать и изменять взаимосвязи типов (классов, интерфейсов, структур, перечней и делегатов), включенных в проект. С помощью этого инструмента можно визуально добавлять, модифицировать и удалять члены типов, а результаты модификации будут отображаться в соответствующем C#-файле. Аналогично, если изменить данный C#-файл, соответствующие изменения будут отражены в окне диаграммы классов.

Для работы с этими возможностями Visual Studio 2005 сначала нужно создать новый файл диаграммы классов. Это можно сделать по-разному, и один из вариантов – щелчок на кнопке View Class Diagram (Просмотр диаграммы классов), которая размещается вверху справа в окне Solution Explorer (рис. 2.22).

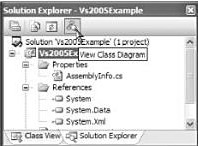


Рис. 2.22. Создание файла диаграммы классов

После этого вы увидите пиктограммы классов, входящих в ваш проект. Щелчок на изображении стрелки будет показывать или скрывать члены типа (см. рис. 2.23).



Рис. 2.23. Просмотр диаграммы классов

Эту утилиту удобно использовать с двумя другими возможностями Visual Studio 2005 – окном Class Details (активизируется из меню View→Other Windows) и разделом Class Designer панели инструментов (активизируется из меню View→Toolboх). Окно Class Details не только показывает структуру выбранного в настоящий момент элемента диаграммы, но и позволяет модифицировать существующие и вставлять новые члены класса (рис. 2.24).

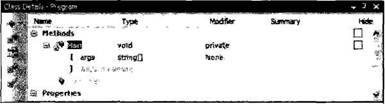


Рис. 2.24. Окно содержимого классов

Панель инструментов (рис. 2.25) позволяет с помощью визуальных средств вставлять в проект новые типы из раздела Class Designer (и создавать связи между этими типами). (Чтобы этот раздел панели инструментов был видимым, окно диаграммы классов должно быть активным.) При этом IDE автоматически создает новые определения C#-типов в фоновом режиме.

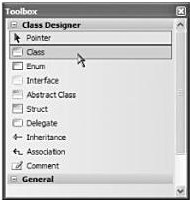


Рис. 2.25. Вставка нового класса с помощью визуальных средств Class Designer

Для примера перетащите новый класс из раздела Class Designer панели инструментов в окно диаграммы классов, В соответствующем диалоговом окне укажите для этого нового класса имя Car (автомобиль). Затем, используя окно Class Details, добавьте в класс открытое строковое поле, назначив, ему имя petName (имя любимца), как показано на рис. 2.26.

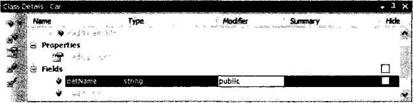


Рис. 2.26. Добавление поля в окно содержимого класса

Если теперь взглянуть на определение C#-класса Car, вы увидите, что оно соответствующим образом обновлено.

public class Car {

 // **Использовать открытые данные без необходимости**

**// не рекомендуется, но здесь это сделано для простоты**

 public string petName;

}

Добавьте в окно диаграммы классов еще один новый класс с именем SportsCar (спортивный автомобиль). Затем в разделе Class Designer панели инструментов выберите пункт Inheritance (Наследование) и щелкните на пиктограмме класса SportsCar. Не отпуская левую кнопку мыши, переместите указатель на пиктограмму класса Car. Если все было сделано правильно, вы должны получить класс SportsCar, являющийся производным класса Car (рис. 2.27).

Чтобы закончить построение примера, добавьте в сгенерированный класс SportsCar открытый метод PrintPetName().

public class SportsCar: Car {

 public void PrintPetName() {

  petName = "Фредди";

  Console.WriteLine("Имя этой машины: {0}", petName);

 }

}

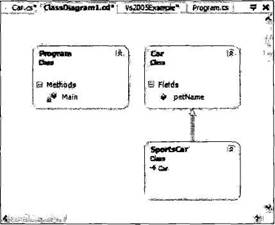


Рис. 2.27. Визуальное получение производного класса из класса, имеющегося в наличии

**Стенд тестирования объектов (ОТВ-тестер)**

Еще одним удобным инструментом визуальный разработки в Visual Studio 2005 является ОТВ-тестер (Object Test Bench – стенд тестирования объектов). Этот инструмент IDE позволяет быстро создать экземпляр класса и выполнить вызов его членов без компиляции и выполнения всего приложения. Это очень удобно в тех случаях, когда вы хотите проверить работу конкретного метода, но в обычных условиях для этого требуется "пройти" через десятки строк программного кода.

Для работы с ОТВ-тестером щелкните правой кнопкой мыши на типе, который вы создаете с помощью окна проектирования классов. Например, щелкните правой кнопкой мыши на типе SportsCar и из появившегося контекстного меню выберите Create Instance→SportsCar(), Появится диалоговое окно, которое позволит задать имя вашей временной объектной переменной (и, если нужно, предоставить конструктору необходимые аргументы). После завершения процесса вы обнаружите свой объект в рамках IDE. Щелкните правой кнопкой мыши на пиктограмме объекта и вызовите метод PrintPetName() (рис. 2.28).

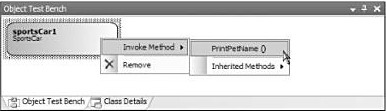


Рис. 2.28. Стенд тестирования объектов в Visual Studio 2005

Вы увидите сообщение "Имя этой машины: Фредди", которое появится в Visual Studio 2005 в рамках консоли Quick.

**Интегрированная справочная система**

В завершение давайте обсудим возможность Visual Studio 2005, которая по определению *должна*быть удобной. Речь здесь идет об интегрированной справочной системе. Документация .NET Framework 2.0 SDK исключительно хороша, очень удобна для чтения и содержит очень много полезной информации. С учетом огромного количества встроенных .NET-типов (их число измеряется тысячами) вы должны быть готовы закатать рукава, чтобы погрузиться в глубины предлагаемой документации. Если же вы к этому не готовы, то при разработке .NET-приложений вы обречены на бесконечные трудности и многочисленные разочарования.

В Visual Studio 2005 предлагается окно Dynamic Help (Динамическая справка), которое (динамически!) изменяет свое содержимое в зависимости от того, какой элемент (окно, меню, ключевое слово исходного кода и т.д.) является активным в настоящий момент. Например, если вы поместите Курсор мыши на класс Console, окно Dynamic Help отобразит набор разделов справки, имеющих отношение к типу System.Console.

Также следует знать об одном очень важном подкаталоге документации .NET Framework 2.0 SDK. В разделе .NET Development→NET Framework SDK→Class Library Reference документации вы найдете полные описания всех пространств имен из библиотек базовых классов .NET (рис. 2.29).

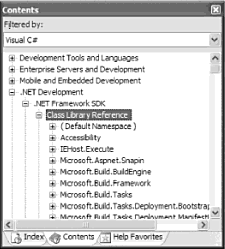


Рис. 2.29. Справка по библиотеке базовых классов .NET

Каждый "узел" здесь определяет набор типов данного пространства имен, члены данного типа и параметры каждого члена. Более того, при просмотре страницы помощи для данного типа сообщается имя компоновочного блока и пространства имен, которые содержат рассматриваемый тип (соответствующая информация размещается в верхней части страницы). Я предполагаю, что при изучении материала данной книги вы доберетесь до очень и очень глубоко спрятанных узлов, чтобы получить важные дополнительные сведения о рассматриваемых объектах.

**Дополнительные средства разработки .NET-приложений**

В заключение хотелось бы обратить ваше внимание на ряд инструментов разработки .NET, которые могут дополнить функциональные возможности выбранной вами IDE. Многие из упомянутых здесь инструментов имеют открытый исходный код, и все они бесплатны. В этой книге нет места для подробного описания этих утилит, но в табл. 2.5 представлены описания инструментов, которые я считаю чрезвычайно полезными, а также URL-адреса, по которым можно найти дополнительную информацию.

**Таблица 2.5.**Подборка средств разработки .NET-приложений

|  |  |  |
| --- | --- | --- |
| **Название** | **Описание** | **URL-адрес** |
| FxCop | Этот инструмент входит в разряд обязательных для любого разработчика  .NET-приложений, заинтересованного в совершенствовании своих программ.  FxCop проверит любой компоновочный блок .NET на соответствие официальным  требованиям и рекомендациям Microsoft .NET | [http://www.gotdotnet.com/tеаm/fxcop](http://www.gotdotnet.com/t%d0%b5%d0%b0m/fxcop) |
| Lab Roeder's Refleсtor для .NET | Этот усовершенствованный декомпилятор/браузер объектов NET позволяет  проанализировать реализацию любого .NET-типа, использующего CIL, C#, Object  Pascal .NET (Delphi) или Visual Basic .NET | [http://www.aisto.сom/roeder/dotfnet](http://www.aisto.xn--om-nmc/roeder/dotfnet) |
| NAnt | NAnt является .NET-эквивалентом Ant – популярного автоматизированного  средства создания модулей Java. NAnt позволяет определять и выполнять  подробные сценарии компоновки, используя синтаксис XML | <http://sourceforge.net/projects/nant> |
| NDoc C | помощью NDoc можно генерировать файлы документации для программного кода  C# (или компилированных компоновочных блоков .NET) в самых популярных  форматах (\*.chm MSDN, XML, HTML, Javаdoc и LaTeX) | <http://sourseforge.net/projects/ndoc> |
| NUnit | NUnit является.NET-эквивалентом инструмента JUnit, предназначенного для  тестирования Java-модулей. С помощью NUnit можно упростить процесс проверки  управляемого программного кода | http://www.nunit.org |
| Vil | Воспринимайте Vil как "старшего брата" разработчика .NET. Этот инструмент  проанализирует программный код .NET и предложит ряд рекомендаций  относительно того, как улучшить его о помощью факторизации,  структурированной обработки исключений и т.д. | http://www.Ibot.com |

Замечание. Функциональные возможности FxCop сейчас интегрированы в Visual Studio 2005. Чтобы в этом убедиться, выполните двойной щелчок на пиктограмме Properties (Свойства) в окне Solution Explorer и активизируйте вкладку Code Analysis (Анализ программного кода).

**Резюме**

Как видите, в ваше полное распоряжение предоставлено множество новых игрушек! Целью этой главы было описание самых популярных средств создания программ на языке C#, которые могут ускорить процесс разработки. Обсуждение началось с описания того, как сгенерировать компоновочный блок .NET, не имея ничего, кроме бесплатного компилятора C# и программы Блокнот. Затем мы рассмотрели приложение TextPad и выяснили, как настроить этот инструмент на редактирование и компиляцию файлов \*.cs с программным кодом.

Были также рассмотрены три интегрированные среды разработки с более широкими возможностями: сначала SharpDevelop с открытым исходным кодом, затем Visual C# 2005 Express и, наконец. Visual Studio 2005 от Microsoft. Эта глава только коснулась всего богатства функциональных возможностей каждого из этих инструментов, чтобы вы могли приступить к самостоятельному изучению выбранной вами среды разработки. В завершение был рассмотрен ряд инструментов разработки .NET с открытым исходным кодом, которые могут предложить разработчику дополнительные возможности.

**ЧАСТЬ II. Язык программирования C#**

**ГЛАВА 3. Основы языка C#**

Воспринимайте эту главу как коллекцию тем, посвященных основным вопросам применения языка C# и использования платформы .NET. В отличие от следующих глав, здесь нет одной ведущей темы, а предлагается иллюстрации целого ряда узких тем, которые вы должны освоить. Это, в частности, типы данных, характеризуемые значениями, и ссылочные типы данных, конструкции условного: выбора и цикла, механизмы приведения к объектному типу и восстановления из "объектного образа", роль System.Object и базовая техника построения классов. По ходу дела вы также узнаете, как в рамках синтаксиса C# обрабатываются строки, массивы, перечни и структуры.

Чтобы иллюстрировать базовые принципы применения языка, мы рассмотрим библиотеки базовых классов .NET и построим ряд примеров приложений, используя различные типы из пространства имен System. В этой главе также рассматривается такая новая возможность языка C# 2005, как тип данных с разрешением принимать значение null. Наконец, вы узнаете, как в C# с помощью ключевого слова namespace объединить типы в отдельное пространство имен.

**Структура простой программы на C#**

Язык C# требует, чтобы вся логика программы содержалась в рамках определения некоторого типа (вспомните из главы 1, что термин *тип* используется для обозначения любого элемента множества {*класс, интерфейс, структура, перечень, делегат*})*.*В отличие от C(++), в C# не позволяется создавать глобальные функций и глобальные элементы данных. В простейшей своей форме программа на C# может быть записана в следующем виде.

**// По соглашению C#-файлы имеют расширение \*.cs.**

using System;

class HelloClass {

 public static int Main(string[] args) {

  Console.WriteLine("Hello World!");

  Console.ReadLine();

  return 0;

 }

}

Здесь определяется тип класса (HelloClass), поддерживающий единственный метод, которому назначено имя Main(). Каждое выполняемое C#-приложение должно содержать класс, определяющий метод Main(), который используется для обозначения точки входа приложения. Как видите, здесь с методом Main() связаны *ключевые*слова public и static. Позже будут представлены их формальные определения, а пока что вам достаточно знать, что открытые члены*(*public) доступны дли других типов, а статические члены (static) рассматриваются на уровне класса (а не на уровне объекта) и поэтому могут вызываться без создания нового экземпляра класса.

**Замечание**. Язык C# является языком, чувствительным к регистру символов. Например, Main здесь отличается от main, а Readlinе – от ReadLine. Поэтому следует подчеркнуть, что все ключевые слова в C# состоят из букв нижнего регистра (public, lock, global и т.д.), а пространства имен, типы, имена членов, а также все интегрированные а них слова начинаются (по соглашению) с прописных букв (например, Console.WriteLine, System.Windows.Forms.MessageBox, System.Data.SqlClient и т.д).

Вдобавок к ключевым словам public и static, этот метод Main() имеет один параметр, который в данном случае является массивом строк (String[] args). В настоящий момент вопрос обработки этого массива мы обсуждать не будем, но следует заметить, что этот параметр может принять любое число аргументов командной строки (вскоре вы узнаете как получить к ним доступ).

Вся программная логика HelloClass содержится в рамках Main(). Здесь используется класс Console, который определен в пространстве имен System. Среди множества других членов там имеется статический элемент WriteLine(), который как вы можете догадаться, посылает текстовую строку на стандартное устройство вывода. Здесь же вызывается Console.ReadLine(), чтобы информация командной строки была видимой в ходе сеанса отладки Visual Studio 2005, пока вы не нажмете клавишу ‹Enter›.

Ввиду того, что здесь метод Main() определен, как метод, возвращающий данные типа integer (целочисленные данные), перед выходом из метода возвращается нуль (означающий успешное завершение). Наконец, как вы можете понять из определения типа HelloClass, в языке C# используется тот вид комментариев, который был принят в C и C++.

**Вариации метода Main()**

Предыдущий вариант Main() был определен с одним параметром (массивом строк) и возвращал данные типа int. Однако это не единственно возможная форма Main(). Для построения точки входа приложения можно использовать любую из следующих сигнатур (в предположении, что она содержится в рамках C#-класса или определения структуры).

**// Возвращаемого типа нет, массив строк в качестве аргумента**

public static void Main(string[] args) {

}

**// Возвращаемого типа нет, аргументов нет.**

public static **void**Main() {

}

**// Возвращаемый тип int (целое), аргументов нет.**

public static **int**Main() {

}

Замечание. Метод Main() можно также определить, как private (частный, приватный), а не public (открытый, общедоступный). Это будет означать, что другие компоновочные блоки не смогут непосредственно вызвать точку входа приложения. В Visual Studio 2005 метод Main() программы автоматически определяется, как приватный.

Очевидно, что при выборе варианта определения Main() нужно учитывать ответы на следующие два вопроса. Bo-первых предполагается ли при выполнении программы обрабатывать предоставленные пользователем параметры командной строки? Если да, то значения параметров должны запоминаться в массиве строк. Во-вторых, нужно ли будет по завершении работы Main() предоставить системе возвращаемое значение? Если да, то возвращаемым типом данных должно быть int, а не void.

**Обработка аргументов командной строки**

Давайте изменим класс HelloClass так. чтобы он мог обрабатывать параметры командной строки.

**// Проверить, передавались ли аргументы командной строки.**

using System;

class HelloClass {

 public static int Main(string[] args) {

  Console.WriteLine("\*\*\* Аргументы командной строки \*\*\*");

  for (int i = 0; i ‹ args.Length; i*++*) Console.WriteLine("Apгyмeнт: {0} ", args[i]);

  …

 }

}

Здесь с помощью свойства Length объект System.Array проверяется, содержит ли массив строк какие-либо элементы (как вы убедитесь в дальнейшем, все массивы в C# на самом деле имеют тип System.Array и таким образом имеют общее множество членов). В результате прохода по всем элементам массива их значения выводятся в окно консоли. Аргументы в командной строке указываются так, как показано на рис*.*3.1.



Рис. 3.1. Аргументы вызова приложения в командной строке

Вместо стандартного цикла for для итераций над массивами входных строк можно использовать ключевое слово C# foreach. Этот элемент синтаксиса будет подробно рассматриваться позже, но вот вам пример его использования:

**// Обратите внимание на то, что при использовании'foreach'**

**// нет необходимости проверять длину массива.**

public static int Main(string[] args) {

 …

**foreach**(string s **in** args) Console.WriteLine("Аргумент: {0} ", s);

 …

}

Наконец, доступ к аргументам командной строки обеспечивает также статический метод GetCommandLineArgs() типа System.Environment. Возвращаемым значением этого метода является массив строк. Его первый элемент идентифицирует каталог, содержащий приложение, а остальные элементы в массиве содержат по отдельности аргументы командной строки (при этом нет необходимости определять для метода Main() параметр в виде массива строк).

public static int Main(string[] args) {

 ...

**// Получение аргументов с помощью System.Environment.**

 string[] theArgs = Environment.GetCommandLineArgs();

 Console.WriteLine("Путь к приложению: {0}", theArgs[0]);

 …

}

**Использование аргументов командной строки в Visual Studio 2005**

Конечный пользователь указывает аргументы командной строки при запуске программы. В процессе разработки приложения вы можете указать флаги командной строки с целью тестирования программы. Чтобы сделать это в Visual Studio 2005, выполните двойной щелчок на пиктограмме Properties (Свойства) в окне Solution Explorer (Обзор решений) и выберите вкладку Debug (Отладка). После этого укажите нужные значения аргументов в поле текста Command line arguments (Аргументы командной строки), рис. 3.2.

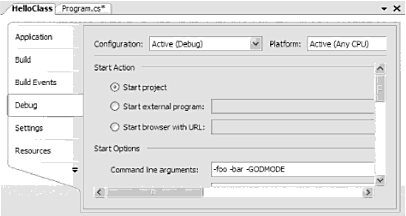


Рис. 3.2. Установка аргументов командной строки в Visual Studio 2005

**Несколько слов о классе System.Environment**

Давайте рассмотрим класс System.Environment подробнее. Этот класс содержит ряд статических членов, позволяющих получить информацию относительно операционной системы, в которой выполняется .NET-приложение. Чтобы иллюстрировать возможности этого класса, измените метод Mаin() в соответствии со следующей логикой.

public static int Main(string[] args) {

 ...

**// Информация об операционной системе.**

 Console.WriteLine("Используемая ОС: {0} ", Environment.OSVersion);

 // **Каталог, в котором находится приложение**.

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

**// Список дисководов на данной машине.**

 string[] drives = Environment.GetLogicalDrives();

 for (int i = 0; i ‹ drives.Length; i*++*)

  Console.WriteLine("Диск {0}: {1} ", i, drives[i]);

**// Версия .NET-платформы, выполняемая на машине.**

 Console.WriteLine("Выполняемая версия .NET: {0} ", Environment.Version);

 …

}

Возможный вариант вывода показан на рис. 3.3.

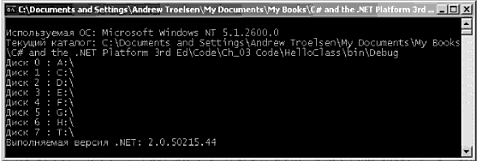


Рис. 3.3. Переменные окружения за работой

Тип System.Envirоnmеnt содержит определения и других членов, а не только представленных в данном примере. В табл. 3.1 показаны некоторые интересные свойства, но непременно загляните в документацию .NET Framework 2.0 SDK, чтобы узнать подробности.

**Таблица 3.1.**Некоторые свойства System.Environment

|  |  |
| --- | --- |
| **Свойстве** | **Описание** |
| MashineName | Имя текущей машины |
| NewLine | Символ перехода на новую строку для текущего окружения |
| ProcessorCount | Число процессоров текущей машины |
| SystemDirectory | Полный путь к системному каталогу |
| UserName | Имя модуля, запустившего данное приложение |

**Определение классов и создание объектов**

Теперь, когда вы знаете о роли Main(), перейдем в задаче построения объектов. Во всех объектно-ориентированных языках делается четкое различие между классами и объектами. Термин *класс*используется для определения пользовательского типа (User-Defined Type – UDT)*,*или, если хотите, *шаблона*. А термин объект применяется для обозначения экземпляра конкретного класса в памяти. Ключевое слово new в C# обеспечивает способ создания объектов. В отличие от других объектно-ориентированных языков (таких как, например, C++), в C# невозможно разместить тип класса в стеке, поэтому если вы попытаетесь использовать переменную класса, которая не была создана с помощью new, вы получите ошибку компиляции. Таким образом, следующий программный код C# оказывается *недопустимым*.

using System;

class HelloClass {

 public static int Main(string[] args) {

**// Ошибка! Используется неинициализированная локальная**

**// переменная. Следует использовать 'new'.**

  HelloClass c1;

  с1.SomeMethod();

  …

 }

}

Чтобы использовать правильные процедуры для создания объектов, внесите следующие изменения.

using System;

class HelloClass {

 public static int Main(string[] args) {

**// Можно объявить и создать объект в одной строке…**

  HelloClass с1 = **new**HelloClass();

**//…или указать объявление и создание в разных строках.**

  HelloClass c2;

  с2 = **new**HelloClass();

  …

 }

}

Ключевое слово new отвечает за вычисление числа байтов, необходимых для заданного объекта, и выделение достаточного объема управляемой динамической памяти (managed heap). В данном случае вы размещаете два объекта типа класса HelloClass. Следует понимать, что объектные переменные C# на самом деле являются *ссылками* на объект в памяти, а не фактическими объектами. Так что c1 и с2 ссылаются на уникальный объект HelloClass, размещенный а управляемой динамической памяти.

**Роль конструкторов**

До сих пор объекты HelloClass строились с помощью *конструктора, заданного по умолчанию,*который, по определению, не имеет аргументов. Каждый класс C# автоматически снабжается типовым конструктором, который вы можете при необходимости переопределить. Этот типовой конструктор используется по умолчанию и гарантирует, что все члены-данные по умолчанию получат подходящие типовые значения (такое поведение характерно для всем конструкторов). Сравните это с ситуацией в C++. где неинициализированные данные указывают на "мусор" (иногда мелочи оказываются очень важными).

Обычно кроме конструктора, заданного по умолчанию, классы предлагают и другие конструкторы. Тем самым вы обеспечиваете возможность инициализации состояния объекта во время его создания, Подобно Java и C++, конструкторы в C# имеют имя, соответствующее имени класса, который они конструируют, и они никогда не возвращают значения (даже значения void). Ниже снова рассматривается тип HelloClass, но с пользовательским конструктором, переопределенным заданным по умолчанию конструктором, и элементом открытых строковых данных.

**// HelloClass c конструкторами.**

class HelloClass {

**// Элемент открытых данных.**

 public string userMessage;

**// Конструктор, заданный по умолчанию.**

 public HelloClass() {

  Console.WriteLine("Вызван конструктор, заданный по умолчанию!");

 }

**// Пользовательский конструктор, связывающий данные состояния**

**// с данными пользователя.**

 public HelloClass(string msg) {

  Console.WriteLine("Вызван пользовательский конструктор!");

  userMessage = msg;

 }

**// точка входа программы.**

 public static int Main(string[] args) {

**// Вызов конструктора, заданного по умолчанию**

  HelloClass c1 = new HelloClass();

  Console.WriteLine("Значение userMessage: {0}\n", c1.userMessage);

**// Вызов параметризованного конструктора**

  HelloClass c2;

  c2 = new HelloClass("Проверка. 1, 2, 3"); Console.WriteLine("Значение userMessage: {0}\n ", c2.userMessage);

  Console.ReadLine();

  return 0;

 }

}

Замечание. Если тип определяет члены (включая и конструкторы) с одинаковыми именами, которые отличаются только числом (или типом) параметров, то соответствующий член называют*перегруженным.*В главе 4 перегрузка будет рассмотрена подробно.

При анализе вывода этой (программы можно заметить что конструктор, заданный по умолчанию, присваивает строковому полю значение (пустое), предусмотренное по умолчанию, в то время как специальный конструктор определяет для члена значение, предоставленное пользователем (pиc. 3.4).

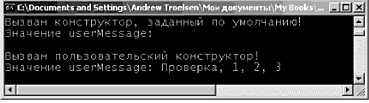


Рис. 3.4. Простая логика конструктора

**Замечание**. После определения пользовательского конструктора для типа класса конструктор, заданный по умолчанию, будет удален. Чтобы в этом случае у пользователей объекта осталась возможность создавать экземпляры типа с помощью конструктора, заданного по умолчанию такой конструктор нужно явно переопределить, как это сделано в предыдущем примере.

**Утечка памяти**

Если вы имеете опыт программирования на языке C++, то у вас в связи с предыдущими примерами программного кода могут возникать вопросы. В частности, следует обратить внимание на то, что метод Main() типа HelloClass не имеет явных операторов уничтожений ссылок c1 и с2.

Это не ужасное упущение, а правило .NET. Как и программистам Visual Basic и Java, программистам C# не требуется уничтожать управляемые объекты явно. Механизм сборки мусора .NET освобождает память автоматически, поэтому в C# не поддерживается ключевое слово delete. В главе 5 процесс сборки мусора будет рассмотрен подробно. До того времени вам достаточно знать лишь о том, что среда выполнения .NET автоматически уничтожит размещенные вами управляемые объекты.

**Определение "объекта приложения"**

В настоящее время тип HelloClass решает две задачи. Во-первых, этот класс определяет точку входа в приложение (метод Main()). Во-вторых, HelloClass поддерживает элемент данных и несколько конструкторов. Все это хорошо и синтаксически правильно, но немного странным может показаться то, что статический метод Main() создает экземпляр того же класса, в котором этот метод определен.

class HelloClass {

 …

 public static int Main(string[] args) {

 HelloClass c1 = new HelloClass();

*…*

 }

}

Такой подход здесь и в других примерах используется только для того, чтобы сосредоточиться на иллюстрации решения соответствующей задачи. Более естественным подходом была бы факторизация типа HelloClass с разделением его на два отдельных класса: HelloClass и HelloApp. При компоновке C#-приложения обычно один тип используется в качестве "объекта приложения" (это тип, определяющий метод Main()), в то время как остальные типы и составляют собственно приложение.

В терминах ООП это называется *разграничение обязанностей.*В сущности, этот принцип проектирования программ требует, чтобы класс отвечал за наименьший объем работы. Поэтому мы можем изменить нашу программу следующим образом (обратите внимание на то, что здесь в класс HelloClass добавляется новый член PrintMessage()).

class HelloClass {

 public string userMessage;

 public HelloClass()  {Console.WriteLine("Вызван конструктор, заданный по умолчанию!");}

 public HelloClass(string msg) {

  Console.WriteLine("Вызван пользовательский конструктор!");

  userMessage = msg;

 }

 public void PrintMessage() {

  Console.WriteLine("Значение userMessage: {0}\n", userMessage);

 }

}

class HelloApp {

 public static int Main(string[] args) {

  HelloClass c1 = new HelloClass("Эй, вы, там…");

  c1.PrintMessage();

 }

}

**Исходный код.**Проект HelloClass размещен в подкаталоге, соответствующем главе 3.

**Класс System.Console**

Многие примеры приложений, созданные для первых глав этой книги, используют класс System.Console. Конечно, интерфейс CUI (Console User Interface – консольный интерфейс пользователя) не так "соблазнителен", как интерфейс Windows или WebUI, но, ограничившись в первых примерах интерфейсом CUI, мы можем сосредоточиться на иллюстрируемых базовых понятиях, не отвлекаясь на сложности построения GUI (Graphical User Interface – графический интерфейс пользователя).

Как следует из его имени, класс Console инкапсулирует элементы обработки потоков ввода, вывода и сообщений об ошибках для консольных приложений. С выходом .NET 2.0 тип Console получил новые функциональные возможности.

В табл. 3.2 представлен список некоторых наиболее интересных из них (но, конечно же, не всех).

**Таблица 3.2.**Подборка членов System.Console, новых для .NET 2.0

|  |  |
| --- | --- |
| **Член** | **Описание** |
| BackgroundColor ForegroundColor | Свойства, устанавливающие цвет изображения/  фона для текущего потока вывода. Могут получать значения  из перечня ConsoleColor |
| BufferHeight BufferWidth | Свойства, контролирующие высоту/ширину буферной  области консоли |
| Clear() | Метод, выполняющий очистку буфера и области  отображения консоли |
| Title | Свойство, устанавливающее заголовок текущей консоли |
| WindowHeight WindowWidth WindowTop WindowLeft | Свойства, контролирующие размеры консоли относительно  заданного буфера |

**Ввод и вывод в классе Console**

Вдобавок к членам, указанным в табл. 3.2, тип Console определяет множество методов, обрабатывающих ввод и вывод, причем все эти методы определены как статические (static), поэтому они вызываются на уровне класса. Вы уже видели, что WriteLine() вставляет текстовую строку (включая символ возврата каретки) в выходной поток. Метод Write() вставляет текст в выходной поток без возврата каретки. Метод ReadLine() позволяет получить информацию из входного потока до символа возврата каретки, a Read() используется дли захвата одного символа из входного потока.

Чтобы проиллюстрировать основные возможности ввода-вывода класса Console, рассмотрим следующий метод Main(), который запрашивает у пользователя некоторую информацию и повторяет каждый элемент в потоке стандартного вывода. На рис 3.5 показан пример выполнения такой программы.

**// Использование класса Console для ввода и вывода.**

static void Main(string[] args) {

**// Эхо для некоторых строк.**

 Console.Write("Введите свое имя: ");

 string s = Console.ReadLine();

 Console.WriteLine("Привет {0} ", s);

 Console.Write("Укажите возpаст: ");

 s = Console.ReadLine();

 Console.WriteLine("Вам {0} год(а)/лет", s);

}



Рис. 3.5. Ввод и вывод с помощью System.Console

**Форматирование консольного вывода**

В этих первых главах вы много раз видели в строковых литералах символы {0}, {1} и др. В .NET вводится новый стиль форматирования строк, немного напоминающий стиль функции printf() в C, но без загадочных флагов %d, %s и %с. Вот простой пример (соответствующий вывод показан на рис. 3.6).

static void Main(string[] args) {

 ...

 int theInt = 90;

 double theDouble = 9.99;

 bool theBool = true;

**// Код '\n' в строковых литералах выполняет вставку**

**// символа перехода на новую строку.**

 Console.WriteLine("Int равно {0}\nDouble равно {1}\nВооl равно {2}", theInt, theDouble, theBool);

}

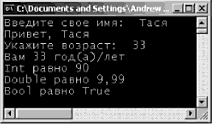


Рис. 3.6. Множество "пустышек" в строковых литералах

Первый параметр метода WriteLine() представляет собой строковый литерал, который содержит опции-заполнители, обозначенные {0}, {1}, {2} и т.д. (нумерация в фигурных скобках всегда начинается с нуля). Остальные параметры WriteLine() являются значениями, которые должны быть вставлены на место соответствующих заполнителей (в данном случае это theInt, theDouble и theBool).

Также следует знать о том, что метод WriteLine() перегружен, чтобы можно было указывать в качестве значения заполнителя массив объектов. Так, строкой формата следующего вида можно представить любое число элементов.

**// Замена заполнителей элементами массива объектов.**

object[] stuff = {"Эй", 20.9, 1, "Там", "83", 99.99933);

Console.WriteLine("Мусор: {0}, {1}, {2}, {3}, {4}, {5}", **stuff);**

Можно также повторять заполнитель в строке. Например, если вы являетесь поклонником Beatles и хотите построить строку "9, Number 9, Number 9", то можете написать следующее.

// Джон говорит,…

Console.WriteLine ("{0}, Number {0}, Number {0}", 9);

**Замечание.**Если имеется несоответствие между числом различных заполнителей в фигурных скобках и числом заполняющих их аргументов, то в среде выполнения генерируется исключение FormatException.

**Флаги форматирования строк .NET**

Если требуется более сложное форматирование, каждый заполнитель может дополнительно содержать различные символы форматирования (в верхнем или в нижнем регистре), как показано в табл. 3.3.

**Таблица 3.3.**Символы форматирования строк .NET

|  |  |
| --- | --- |
| **Символы форматирования строк** | **Описание** |
| C или с | Используются для форматирования денежных значений. По умолчанию перед этим  флагом будет размещаться символ локальной денежкой единицы (скажем, знак  доллара [$] для U.S. English) |
| D или d | Используются для форматирования десятичных чисел. Этот флаг также указывает  минимальное число знаков, используемое для представления значения |
| Е или е | Используются для представлений в экспоненциальном формате |
| F или f | Используются для представления в формате с фиксированным разделителем |
| G или g | Обозначают *general*(общий [формат]). Эти символы можно использовать для  представления чисел в формате с фиксированным разделителем или в  экспоненциальном формате |
| N или n | Используются для базового числового форматирования (с разделением групп  разрядов) |
| X или x | Используются для представления в шестнадцатиричном формате. Если используется  X (в верхнем регистре), то в шестнадцатиричном представлении используются  символы верхнего регистра |

Символы форматирования добавляются в виде суффикса к соответствующему заполнителю через двоеточие (например, {0:C}, {1:d}, {2:X} и т.д.). Предположим, что вы добавили в Main() следующий программный код.

**// Используем некоторые дескрипторы формата.**

static void Main(string[] args) {

 …

 Console.WriteLine("Формат C: **{0:C}**", 99989.987);

 Console.WriteLine("Формат D9: **{0:D9}**", 99999);

 Console.WriteLine("Формат E: **{0:E}**", 99999.76543);

 Console.WriteLine("Формат F3: **{0:F3}**"**,**99999.9999);

 Console.WriteLine("Формат N: **{0:N}**", 99999);

 Console.WriteLine("Формат X: **{0:X}**", 99999);

 Console.WriteLine("Фopмaт x: **{0:x}**"**,**99999)**;**

}

Использование символов форматирования в .NET не ограничивается консольными приложениями. Те же флаги можно использовать в контексте статического метода String.Format(). Это может быть полезно тогда, когда в памяти нужно построить строку с числовыми значениями, подходящую для использования в приложениях любого типа (Windows Forms, ASP.NET, Web-сервисы XML и т.д.).

static void Main(string[] args) {

**// Использование статического метода String.Format()**

**// для построения новой строки.**

 string formatStr;

 formatStr = String.Format("Хотите получить {0:C} на свой счет?", 99989.987);

 Console.WriteLine(formatStr);

}

На рис. 3.7 показан пример вывода данной программы.

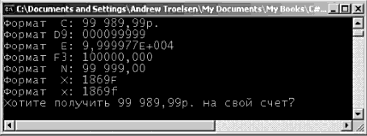


Рис. 3.7. Флаги форматирования строк в действии

**Исходный код.**Проект BasicConsoleIO размещен в подкаталоге, соответствующем главе 3.

**Доступность членов**

Прежде чем двигаться дальше, мы должны обсудить вопрос доступности, или "видимости" членов. Члены (методы, поля, конструкторы и т.д.) данного класса или структуры должны указать свой уровень доступности. Если член определяется без указания ключевого слова, характеризующего доступность, этот член по умолчанию определяется как private. В C# используются модификаторы доступности методов, перечисленные в табл. 3.4.

Вы, наверное, уже знаете, что доступ к открытым членам можно получить с помощью объектной ссылки, используя операцию, обозначаемую точкой (.). Приватные члены недоступны извне по объектной ссылке, но могут вызываться объектами внутри, чтобы экземпляр мог выполнить свою работу (т.е. это частные вспомогательные функции).

Таблица 3.4. Ключевые слова C#, указывающие уровень доступности

|  |  |
| --- | --- |
| **Модификатор доступности C#** | **Описание** |
| public (открытый, общедоступный) | Помечает метод, как доступный из объектной переменной,  а также из всех производных классов |
| private (частный, приватный) | Помечает метод, как доступный только из класса  , определяющего этот метод. В C# любой член по умолчанию  определяется, как private |
| protected (защищенный) | Помечает метод, как доступный для определяющего класса,  а также для любого производного класса. Однако защищенные  методы не доступны из объектной переменной |
| internal (внутренний) | Определяет метод, как доступный для любого типа только  внутри данного компоновочного блока, но не снаружи |
| protected internal (внутренний защищенный) | Определяет метод, доступ к которому ограничивается рамками  текущего компоновочного блока или типами, созданными из  определяющего класса в данном компоновочном блоке |

Защищенные члены оказываются полезными только при создании иерархии классов, что будет темой обсуждения главы 4. Что касается внутренних, и защищённых членов, то они обычно используются при создании библиотек программного кода .NET (например, управляемых библиотек \*.dll, что будет обсуждаться в главе 11).

Чтобы проиллюстрировать применение указанных ключевых слов, создадим класс (SomeClass), в котором используются все указанные модификаторы доступности членов.

**// Опции доступности членов.**

class SomeClass {

 // **Доступен везде.**

 public void PublicMethod() {}

 // **Доступен только из типов SomeClass.**

 private void PrivateMethod() {}

 // **Доступен из SomeClass и потомков**.

 protected void ProtectedMethod() {}

 // **Доступен только в рамках данного компоновочного блока**.

 internal void InternalMethod() {}

 // **Защищенный доступ внутри компоновочного блока**.

 protected internal void ProtectedInternalMethod() {}

 // **В C# при отсутствии явных указаний**

**// члены по умолчанию считаются приватными.**

 void SomeMethod(){}

}

Теперь, создав экземпляр класса SomeClass, попытаемся вызвать каждый из его методов, используя операцию, обозначаемую точкой.

static void Main(string[] args) {

**// Создается объект и выполняется попытка вызова членов.**

 SomeClass с = new SomeClass();

 c.PublicMethod();

 с.InternalMethod();

 с.ProtectedInternalMethod();

 с.PrivateMethod();**// Ошибка!**

 с.ProtectedMethod();**//Ошибка!**

 с.SomeMethod();**// Ошибка!**

}

Если скомпилировать эту программу, вы обнаружите, что защищенные и частные члены вне объекта *не*доступны.

**Исходный код.**Проект MemberAccess размещен в подкаталоге, соответствующем главе 3.

**Доступность типов**

Типы (классы, интерфейсы, структуры, перечни и делегаты) также могут использовать модификаторы доступности, но только public или internal. Когда вы создаете общедоступный тип (public), то гарантируете, что он будет доступным для других типов как в текущем компоновочном блоке, так и во внешних компоновочных блоках. Это может оказаться полезным только тогда, когда вы создаете библиотеку программного кода (см. главу 11), но здесь мы можем привести пример использования этого модификатора доступности,

**// Этот тип может использоваться любым компоновочным блоком.**

public class MyClass()

Внутренний (internal) тип, с другой стороны, может использоваться только компоновочным блоком, в котором этот тип определен. Если создать библиотеку программного кода .NET, в которой будут определены внутренние типы, то компоновочные блоки, ссылающиеся на эту библиотеку (файл \*.dll), не смогут увидеть эти типы, создать их экземпляры или как-то иначе взаимодействовать с ними.

Характеристикой доступности, принимаемой по умолчанию для типов в C#, является internal, поэтому если вы не укажете явно ключевое слово public, то в результате будет создан внутренний тип.

**// Эти классы могут использоваться только внутри**

**// текущего компоновочного блока.**

internal class MyHelperClass{}

class FinalHelperClass{} //По умолчанию тип будет внутренним.

**Замечание.**В главе 4 будет говориться о вложенных типах. Вы узнаете, что вложенные типы тоже могут быть объявлены, как приватные.

**Значения, назначаемые переменным по умолчанию**

Членам-переменным классов автоматически присваиваются подходящие значения, предусмотренные по умолчанию. Эти значения для каждого типа данных, свои, но правила их выбора достаточно просты:

• для типа bool устанавливается значение false;

• числовым данным присваивается значение 0 (или 0.0, если это данные с плавающим разделителем);

• для типа string устанавливается значение null;

• для типа char устанавливается значение '\0';

• для ссылочных типов устанавливается значение null.

С учетом этих правил проанализируйте следующий программный код.

**// Поля типа класса получают значения по умолчанию.**

class Test {

 public int myInt; **// Устанавливается равным 0.**

 public string myString; **// Устанавливается равным null.**

 public bool myBool; **// Устанавливается равным false.**

 public object myObj; **// Устанавливается равным null.**

}

**Значения, назначаемые по умолчанию, и локальные переменные**

Совсем по-другому обстоит дело тогда, когда объявляются локальные переменные, видимые в пределах данного члена. При определении локальной переменной вы *должны*назначить ей начальное значение, прежде чем начать ее использование, поскольку такая перемен наяне не получает начального значения по умолчанию. Например, следующий программный код приведет к ошибке компиляции,

**// Ошибка компиляции! Переменная 'localInt' должна получить**

**// начальное значение до ее использования.**

static void Main(string[] args) {

 int localInt;

 Console.WriteLine(localInt);

}

Исправить проблему очень просто. Следует присвоить переменной начальное значение.

**// Так лучше: теперь все довольны.**

static void Main(string[] args) {

**int localInt**= **0;**

 Console.WriteLine(localInt);

}

**Замечание.**Правило обязательного присваивания начальных значений локальным переменным имеет одно исключение: если переменная используется в качестве *выходного параметра*(это понятие будет рассмотрено немного позже), то устанавливать начальное значение такой переменной не требуется.

**Синтаксис инициализации членов-переменных**

Типы класса обычно имеют множество членов-переменных (также называемых *полями*). Если в классе можно определять множество конструкторов, то может возникнуть не слишком радующая программиста необходимость многократной записи одного и того же программного кода инициализации для каждой новой реализации конструктора. Это вполне реально, например, в том случае, когда вы не хотите принимать значение члена, предусмотренное по умолчанию. Так, чтобы член-переменная (myInt) целочисленного типа всегда инициализировался значением 9, вы можете записать следующее.

**// Все это хорошо, но такая избыточность…**

class Test {

 public int myInt;

 public string myString;

 public Test() {myInt = 9;}

 public Test(string s) {

  myInt = 9;

  myString = s;

 }

}

Альтернативой может быть определение вспомогательной функции, вызываемой всеми конструкторами. При этом уменьшается общее число повторений для операции *присваивания,*но теперь возникает следующая избыточность,

**// Все равно остается избыточность…**

class Test {

 public int myInt;

 public string myString;

 public Test() {InitData();}

 public Test(string s) {

  myString = s;

  InitData();

 }

 private void InitData() {myInt = 9;}

}

Оба эта подхода вполне легитимны, но в C# позволяется назначать членам типа начальные значения в рамках деклараций (вы, наверное, знаете, что другие объектно-ориентированные языки [например, C++], не позволяют такую инициализацию членов). В следующем фрагменте программного кода обратите внимание на то, что инициализация может выполняться и для внутренних объектных ссылок, а не только для числовых типов данных.

**// Если нужно отказаться от значений, предусмотренных по умолчанию,**

**// эта техника позволяет избежать повторной записи программного**

**// хода инициализации в каждом конструкторе.**

class Test {

 public int myInt = 9;

 public string myStr = "Мое начальное значение. ";

 public SportsCar viper = new SportsCar(Color.Red);

 ...

}

**Замечание**. Инициализация членов выполняется до выполнения программной логики конструктора. Если присвоить значение полю в самом конструкторе, это сведет на нет инициализацию члена.

**Определение констант**

Итак, вы знаете, как объявить переменные класса. Теперь давайте выясним, как определить данные, изменить которые не предполагается. Для определения переменных с фиксированным, неизменяемым значением в C# предлагается ключевое слово const. После определения значения константы любая попытка изменить это значение приводит к ошибке компиляции. В отличие От C++, в C# ключевое слово const нельзя указывать для параметров и возвращаемых значений – оно предназначено для создания локальных данных и данных уровня экземпляра.

Важно понимать, что значение, присвоенное константе, во *время компиляции*уже должно быть известно, поэтому константу нельзя инициализировать объектной ссылкой (значение последней вычисляется в среде выполнения). Чтобы проиллюстрировать использование ключевого слова const, рассмотрим следующий тип класса.

class ConstData {

**// Значение, присваиваемое константе, должно быть известно во время компиляции.**

 public const string BestNbaTeam = "Timberwolves";

 public const double SimplePI = 3.14;

 public const bool Truth = true;

 public const bool Falsity = !Truth;

}

Обратите внимание на то, что значения всех констант известны во время компиляции. И действительно, если просмотреть эти константы с помощью ildasm.exe, то вы обнаружите, что их значения будут "жестко" вписаны в компоновочный блок, как показано на рис. 3.8. (Ничего более постоянного получить невозможно!)

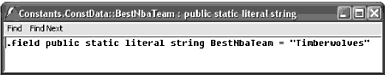


Рис. 3.8. Ключевое слово const вписывает "свое" значение прямо в метаданные компоновочного блока

**Ссылки на константы**

Если нужно сослаться на константу, определенную внешним типом, вы должны добавить префикс имени типа (например, ConstData.Truth), поскольку поля-константы являются *неявно статическими.*Однако при ссылке на константу, определенную в рамках текущего типа (или в рамках текущего члена), указывать префикс имени типа не требуется. Чтобы пояснить это, рассмотрим следующий класс.

class Program {

 public const string BestNhlTeam = "Wild";

 static void Main(string[] args) {

  // **Печать значений констант, определенных другими типами.**

  Console.WriteLine("Константа Nba: {0}", ConstData.BestNbaTeam);

  Console.WriteLine("Константа SimplePI: {0}", ConstData.SimplePI);

  Console.WriteLine("Константа Truth: {0}", ConstData.Truth);

  Console.WriteLine("Константа Falsity: {0}", ConstData.Falsity);

  // **Печать значений констант члена.**

  Console.WriteLine("Константа Nhl: {0}", BestNhlTeam);

  // **Печать значений констант локального уровня.**

  const int LocalFixedValue = 4;

  Console.WriteLine("Константа Local: {0}", LocalFixedValue);

  Console.ReadLine();

 }

}

Обратите внимание на то, что для доступа к константам класса ConstData необходимо указать имя типа. Однако класс Program имеет прямой доступ к константе BestNhlTeam, поскольку она была определена в пределах собственной области видимости класса. Константа LocalFixedValue, определенная в Main(), конечно же, должна быть доступной только из метода Main().

**Исходный код.**Проект Constants размещен в подкаталоге, соответствующем главе 3.

**Определение полей только для чтения**

Как упоминалось выше, значение, присваиваемое константе, должно быть известно во время компиляции. Но что делать, если нужно создать неизменяемое поле, начальное значение которого будет известно только в *среде выполнения?*Предположим, что вы создали класс Tire (покрышка), в котором обрабатывается значение ID (идентификатор) производителя. Кроме того, предположим, что вы хотите сконфигурировать этот тип класса так, чтобы в нем поддерживалась пара известных экземпляров Tire, чьи значения не должны изменяться. Если использовать ключевое слово const, вы получите ошибку компиляции, поскольку адрес объекта в памяти становится известным только *в среде выполнения*.

class Tire {

**// Поскольку адреса объектов определяются в среде выполнения,**

**// здесь нельзя использовать ключевое слово 'const.**'!

 public const Tire Goodstone = new Tire(90); // **Ошибка!**

 public const Tire FireYear = new Tire(100); // **Ошибка!**

 public int manufactureID;

 public Tire() {}

 public Tire(int ID) { manufactureID = ID;}

}

Поля, доступные только для чтения, позволяют создавать элементы данных, значения которых остаются неизвестными в процессе трансляции, но о которых известно, что они никогда не будут изменяться после их создания. Чтобы определить поле, доступное только для чтения, используйте ключевое слово C# readonly.

class Tire {

 public **readonly**Tire GoodStone = new Tire(90);

 public **readonly**Tire FireYear = new Tire(100);

 public int manufactureID;

 public Tire() {}

 public Tire (int ID) {manufactureID = ID;}

}

С такой модификацией вы сможете не только выполнить компиляцию, но и гарантировать, что при изменении значений полей GoodStone и FireYear в программе вы получите сообщение об ошибке.

static void Main(string[] args) {

**// Ошибка!**

**// Нельзя изменять значение поля, доступного только для чтения.**

 Tire t = new Tire();

 t.FireYear = new Tire(33);

}

Поля, доступные только для чтения, отличаются от констант еще и тем, что таким полям можно присваивать значения в контексте конструктора. Это может оказаться очень полезным тогда, когда значение, которое нужно присвоить доступному только для чтения полю, считывается из внешнего источника (например, из текстового файла или из базы данных). Рассмотрим другой класс, Employee (служащие), который определяет доступную только для чтения строку, изображающую SSN (Social Security Number – номер социальной страховки в США). Чтобы обеспечить пользователю объекта возможность указать это значение, можно использовать следующий вариант программного кода.

class Employee {

 public readonly string SSN;

 public Employee(string empSSN) {

  SSN = empSSN;

 }

}

Здесь SSN является значением readonly (только для чтения), поэтому любая попытка изменить это значение вне конструктора приведет к ошибке компиляции.

static void Main(string[] args) {

 Employee e = new Employee("111-22-1111");

 e.SSN = "222-22-2222"; // **Ошибка!**

}

**Статические поля только для чтения**

В отличие от данных-констант, доступные только для чтения поля не причисляются автоматически к группе статических. Если вы хотите использовать значения доступных только для чтения полей на уровне классов, используйте ключевое слово static.

class Tire {

 public **static**readonly Tire GoodStone = new Tire(90);

 public **static**readonly Tire FireYear = new Tire(100);

 ...

}

Вот пример использования нового типа Tire.

static void Main(string[] args) {

 Tire myTire = Tire.FireYear;

 Console.WriteLine("Код ID моих шин: {0}", myTire.manufactureID);

}

**Исходный код.**Проект ReadOnlyFields размещен в подкаталоге, соответствующем главе 3.

**Ключевое слово static**

Как уже говорилось в этой главе, члены классов (и структур) в C# могут определяться с ключевым cловом static В этом случае соответствующий член должен вызываться непосредственно на уровне класса, а не экземпляра типа. Для иллюстрации рассмотрим "знакомый" тип System.Console. Вы уже могли убедиться, что метод WriteLine() вызывается не с объектного уровня.

**// Ошибка! WriteLine() – это не метод уровня экземпляра!**

Console с = new Console();

c.WriteLine ("Так печатать я не могу…");

Вместо этого нужно просто добавить префикс имени типа к имени статического члена WriteLine().

**// Правильно! WriteLine() – это статический метод.**

Console.WriteLine("Спасибо…");

Можно сказать, что статические члены являются элементами, которые (до мнению разработчика типа) оказываются "слишком банальными", чтобы создавать для них экземпляры типа. При создании типа класса вы можете определить любое число статических членов и/или членов уровня экземпляра.

**Статические методы**

Рассмотрим следующий класс Teenager (подросток), который определяет статический метод Complain(), возвращающий случайную строку, полученную с помощью вызова частной вспомогательной функции GetRandomNumber().

class Teenager {

 private **static** Random r = new Random();

 private **static** int GetRandomNumber(short upperLimit) { return r.Next(upperLimit);}

 public **static** string Complain() {

  string[] messages = new string [5] {"А почему я?", "Он первый начал!", "Я так устал…", "Ненавижу школу!", "Это нечестно!"};

  return messages[GetRandomNumber(5)];

 }

}

Обратите внимание на то, что член-переменная System.Random и метод GetRandomNumber(), определяющий вспомогательную функцию, также o6ъявлeны как статические члены класса Teenager, согласно правилу, по которому статические члены могут оперировать *только* статическими членами.

**Замечание**. Позвольте мне повториться. *Статические члены могут воздействовать только на статические члены.*Если в статическом методе вы попытаетесь использовать нестатические члены (также называемые *данными уровня экземпляра*), вы получите ошибку компиляции.

Как и в случае любого другого статического члена, чтобы вызвать Complain(), следует добавить префикс имени класса.

// Вызов статического метода Complain класса Teenager

static void Main(string[] args) {

 for (int i = 0; i ‹ 10; i++) Console.WriteLine("-› {0}", **Teenager.Complain()**);

}

И, как и в случае любого нестатического метода, если бы метод Complain() не был обозначен, как static, нужно было бы создать экземпляр класса Teenager, чтобы вы могли узнать о проблеме дня.

**// Нестатические данные должна вызываться на объектном уровне.**

Teenager joe = new Teenager();

joe.Complain();

**Исходный код.**Проект StaticMethods размещен в подкаталоге, соответствующем главе 3.

**Статические данные**

Вдобавок к статическим методам, тип может также определять статические данные (например, член-переменная Random в предыдущем классе Teenager). Следует понимать, что когда класс определяет нестатические данные, каждый объект данного типа поддерживает приватную копию соответствующего поля, Рассмотрим, например, класс, который моделирует депозитный счет,

**// Этот класс имеет элемент нестатических данных.**

class SavingsAccount {

 public double сurrBalance;

 public SavingsAccount(double balance) {сurrBalance = balance;}

}

При создании объектов SavingsAccount память для поля сurrBalance выделяется для каждого экземпляра. Статические данные, напротив, размещаются один раз и совместно используются всеми экземплярами объекта данного типа. Чтобы привести пример применения статических данных, добавим в класс SavingsAccount элемент currInterestRate.

class SavingsAccount {

 public double currBalance;

 public **static**double currInterestRate = 0.04;

 public SavingsAccount(double balance) { currBalance = balance; }

}

Если теперь создать три экземпляра SavingsAccount, как показано ниже

static void Main(string[] args) {

 // **Каждый объект SavingsAccount имеет свою копию поля currBalance.**

 SavingsAccount s1 = new SavingsAccount (50);

 SavingsAccount s2 = new SavingsAccount(100);

 SavingsAccount s3 = new SavingsAccount(10000.75);

}

то размещение данных в памяти должно быть примерно таким, как показано на рис. 3.9.

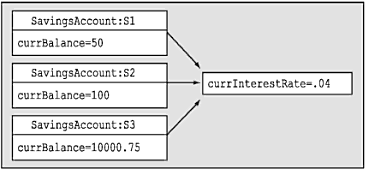


Рис. 3.9. Статические данные совместно используются всеми экземплярами определяющего их класса

Давайте обновим класс SavingsAccount, определив два статических метода для получения и установки значений процентной ставки. Как была замечено выше, статические методы могут работать только со статическими данными. Однако нестатический метод может использовать как статические, так и нестатические данные. Это имеет смысл, поскольку статические данные доступны всем экземплярам типа. С учетом этого давайте добавим еще два метода уровня экземпляра, которые будут использовать переменную процентной ставки.

class SavingsAccount {

 public double currBalance;

 public static double currInterestRate = 0.04;

 public SavingsAccount(double balance) { currBalance balance;}

 // **Статические методы получения/установки процентной ставки.**

 public **static**void SetInterestRate(double newRate) { currInterestRate = newRate; }

 public **static**double GetInterestRate() { return currInterestRate; }

 // **Методы экземпляра получения/установки текущей процентной ставки.**

 public void SetInterestRateObj(double newRate) { currInterestRate = newRate; }

 public double GetInterestRateObj() { return currInterestRate; }

}

Теперь рассмотрим следующий вариант использования этого класса и соответствующий вывод, показанный на рис. 3.10.

static void Main(string [] args) {

 Console.WriteLine("\*\*\* Забавы со статическими данными \*\*\*");

 SavingsAccount s1 = new SavingsAccount(50);

 SavingsAccount s2 = new SavingsAccount(100);

 // **Получение и установка процентной ставки.**

 Console.WriteLine("Процентная ставка: {0}", s1.GetInterestRateObj());

 s2.SetInterestRateObj(0.08);

**// Создание нового объекта.**

**// Это НЕ 'переустанавливает' процентную ставку.**

 SavingsAccount s3 = new SavingsAccount(10000.75);

 Console.WriteLine("Процентная ставка: {0}", SavingsAccount.GetlnterestRate());

 Console.ReadLine();

}

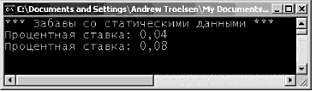


Рис. 3.10. Статические данные размещаются один раз

**Статические конструкторы**

Вы уже знаете о том, что конструкторы используются для установки значения данных типа во время его создания, Если указать присваивание значения элементу статических данных в рамках конструктора уровня экземпляра, вы обнаружите, что это значение переустанавливается каждый раз при создании нового объекта! Например, изменим класс SavingsAccount так.

class SavingsAccount {

 public double currBalance;

 public static double currInterestRate;

 public SavingsAccount(double balance) {

  currBalance = balance;

**currInterestRate = 0.04;**

 }

}

Если теперь выполнить предыдущий метод Main(), вы увидите совсем другой вывод (рис. 3.11). Обратите внимание на то, что в данном случае переменная currInterestRate переустанавливается каждый раз при создании нового объекта SavingsAccount.



Рис. 3.11. Присваивание значений статическим данным в конструкторе "переустанавливает" эти значения

Вы всегда можете установить начальное значение статических данных, используя синтаксис инициализации члена, но что делать, если значение для статических данных нужно получить из базы данных или внешнего файла? Для решения таких задач требуется, чтобы в контексте метода можно было использовать соответствующие операторы программного кода. По этой причине в C# допускается определение *статического конструктора.*

class SavingsAccount {

 …

**// Статический конструктор.**

 static SavingsAccount() {

  Console.WriteLine("В статическом конструкторе.");

  currInterestRate = 0.04;

 }

}

Вот несколько интересных замечаний, касающихся статических конструкторов.

• Любой класс (или структура) может определять только один статический конструктор.

• Статический конструктор выполняется только один раз, независимо от того, сколько создается объектов данного типа.

• Статический конструктор не может иметь модификаторов доступности и параметров.

• Среда выполнения вызывает статический конструктор, когда создается экземпляр класса, или перед тем, как получить доступ к первому вызываемому статическому члену.

• Статический конструктор выполняется до выполнения любого конструктора уровня экземпляра.

Теперь значение статических данных при создании новых объектов SavingsAccount сохраняется, и соответствующий вывод будет идентичен показанному на рис. 3.10.

**Статические классы**

Язык C# 2005 расширил область применения ключевого слова static путем введения в рассмотрение *статических классов.*Когда класс определен, как статический, он не допускает создания экземпляров с помощью ключевого слова new и может содержать только статические члены или поля (если это условие не будет выполнено, вы получите ошибку компиляции).

На первый взгляд может показаться, что класс, для которого нельзя создать экземпляр, должен быть совершенно *бесполезным.*Однако если вы создаете класс, не содержащий ничего, кроме статических членов и/или констант, то нет никакой необходимости и в локализации такого класса. Рассмотрим следующий тип.

**// Статические классы могут содержать только**

**// статические члены и поля-константы.**

static class UtilityClass {

 public static void PrintTime() { Console.WriteLine(DateTime.Now.ToShortTimeString());}

 public static void PrintDate() {Console.WriteLine(DateTime.Today.ToShortDateString());}

}

При наличии модификатора static пользователи объекта не смогут создавать экземпляры UtilityClass.

static void Main(string[] args) {

 UtilityClass.PrintDate();

**// Ошибка компиляции!**

**// Нельзя создавать экземпляры статических классов.**

 UtilityClass u = new UtilityClass();

 …

}

До появления C# 2005 единственной возможностью для запрета на создание таких типов пользователями объекта было или переопределение конструктора, заданного по умолчанию, как приватного, или обозначение класса, как абстрактного типа, с помощью ключевого слова C# abstract (подробно абстрактные типы обсуждаются в главе 4).

class UtilityClass {

**private**UtilityClass(){}

 …

}

**abstract**class UtilityClass {

 …

}

Эти конструкции по-прежнему доступны, но с точки зрения типовой безопасности использование статических классов является более выгодным решением, поскольку указанные выше варианты не исключают присутствия нестатических членов в определении класса.

**Исходный код.**Проект StaticData размещен в подкаталоге, соответствующем главе 3.

**Модификаторы параметров методов**

Методы (и статические, и уровня экземпляра) могут использовать параметры, передаваемые вызывающей стороной. Однако, в отличие от некоторых других языков программировании, в C# предлагается множество модификаторов параметров, которые контролируют способ передачи (и, возможно, возврата) аргументов для данного метода, как показано в табл. 3.5,

Таблица 3.5. Модификаторы параметров C#

|  |  |
| --- | --- |
| **Модификатор параметров** | **Описание** |
| (нет) | Если параметр не помечен модификатором, то предполагается передача параметра  *по значению,*т.е. вызываемый метод получит копию оригинальных данных |
| out | Выходные параметры устанавливаются вызываемым методом (и, таким образом,  передаются по ссылке). Если вызываемый метод не назначит выходные параметры,  генерируется ошибка компиляции |
| params | Позволяет переслать произвольное число аргументов одинакового типа в виде единого параметра. Для любого метода допускается только один модификатор params и только для последнего параметра метода |
| ref | Соответствующее значение задается вызывающей стороной, но вызываемый метод  *может*это значение изменить (поскольку данные передаются по ссылке).  Если вызываемый метод не присвоит значение параметру ref, ошибка компиляции не  генерируется |

**Способ передачи параметров, используемый по умолчанию**

По умолчанию параметр передается в функцию *по значению.*Попросту говоря, если не определить для аргумента модификатор, то в функцию передаётся копия переменной.

**// По умолчанию аргументы передаются по значению.**

public static int Add(int x, int y) {

 int ans = x + y;

**// Вызывающая сторона не увидит этих изменений,**

**// поскольку модифицируется копия оригинальных данных.**

 x = 10000; у = 88888;

 return ans;

}

Здесь входные целочисленные параметры передаются по значению. Поэтому, если изменить значения параметров внутри данного метода, то вызывающая сторона об этом не узнает, поскольку изменяются значения копий целочисленных данных вызывающего объекта.

static void Main(string[] args) {

 int x = 9, y = 10;

 Console.WriteLine ("До вызова: X: {0}, Y: {1}", x, y);

 Console.WriteLine("Ответ: {0}", Add(x, y));

 Console.WriteLine("После вызова: X: {0}, Y: {1}", x, у);

}

Как вы и должны ожидать, значения х и у остаются теми же и после вызова Add().

**Модификатор out**

Теперь рассмотрим использование параметров out (от output – выходной). Если метод определен с выходными параметрами, то необходимо назначить этим параметрам подходящие значения до выхода из метода (если этого не сделать, будет сгенерирована ошибка компиляции).

Ниже для иллюстрации предлагается альтернативный вариант метода Add(), использующий C#-модификатор out и возвращающий сумму двух целых чисел в виде выходного параметра (обратите внимание на то, что возвращаемым значением самого метода теперь будет void).

**// Выходные параметры задаются членом.**

public static **void** Add(int x, int y, **out** int ans) {

 ans = x + y;

}

При вызове метода с выходными параметрами тоже требуется указать модификатор out. Локальным переменным, используемым в качестве выходного параметра, не требуется присваивать значения до их использования (эти значения после вызова все равно будут потеряны), Например:

static void Main(string[] args) {

 // **Нет необходимости задавать значения**

**// локальным выходным переменным.**

 int ans;

 Add(90, 90, **out**ans);

 Console.WriteLine("90 + 90 = {0} ", ans);

}

Этот пример предлагается здесь только для иллюстрации: нет никакой необходимости возвращать значение суммы с помощью выходного параметра. Но сам модификатор out играет очень важную роль: он позволяет вызывающей стороне получить множество возвращаемых значений от одного вызова метода.

**// Возвращение множества выходных параметров.**

public static void FillTheseVals(out int a, out string b, out bool c) {

 а = 9;

 b = "Радуйтесь своей строке.";

 с = true;

}

Вызывающая сторона может вызвать этот метод следующим образом.

static void Main(string[] args) {

 int i; string str; bool b;

 FillTheseVals(**out** i, **out** str, **out** b);

 Console.WriteLine("Int равно: {0}", i);

 Console.WriteLine("String равно: (0}", str);

 Console.WriteLine("Boolean равно: {0}", b);

}

**Модификатор ref**

Теперь рассмотрим, использование в C# модификатора ref (от reference – ссылочный). Ссылочные параметры нужны тогда, когда требуется позволить методу изменять данные, объявленные в контексте вызова (например, в функциях сортировки или обмена данными). Обратите внимание на различие между выходными и ссылочными параметрами.

• Выходные параметры не требуется инициализировать перед передачей их методу. Причина в том, что сам метод должен присвоить значения выходным параметрам.

• Ссылочные параметры *необходимо*инициализировать до того, как они будут переданы методу. Причина в том, что передается ссылка на существующую переменную. Если не присвоить переменной начальное значение, это будет означать использование неинициализированной переменной.

Давайте продемонстрируем использование ключевого слова ref с помощью метода, в котором осуществляется обмен значениями двух строк.

**// Ссылочные параметры.**

public static void SwapStrings(ref string s1*,*ref string s2) {

 string tempStr = s1;

 s1 = s2;

 s2 = tempStr;

}

Этот метод можно вызвать так.

static void Main(string[] args) {

 string s = "Первая строка";

 string s2 = "Вторая строка";

 Console.WriteLine("До: {0}, {1} ", s, s2);

 SwapStrings(**ref**s, **ref**s2);

 Console.WriteLine("После: {0}, {1} "*,*s, s2);

}

Здесь вызывающая сторона присваивает начальное значение локальным строковым данным (s и s2). По завершении вызова SwapStrings() строка s содержит значение "Вторая строка", a s2 – значение "Первая строка".

**Модификатор params**

Нам осталось рассмотреть модификатор params, позволяющий создавать методы, которым можно направить множество однотипных аргументов в виде *одного параметра.*Чтобы прояснить суть дела, рассмотрим метод, возвращающий среднее для любого числа значений,

**// Возвращение среднего для 'некоторого числа' значений.**

static double CalculateAverage(params double[] values) {

 double sum = 0;

 for (int i = 0; i ‹ values.Length; i++) sum += values[i];

 return (sum / values.Length);

}

Этот метод принимает массив параметров, состоящий из значений с двойной точностью. Метод фактически говорит следующее: "Дайте мне *любой набор значений*с *двойной точностью,*и я вычислю для них среднюю величину". Зная это, вы можете вызвать CalculateAverage() одним из следующих способов (если не использовать модификатор params в определении CalculateAverage(), то первый из указанных ниже вариантов вызова этого метода должен привести к ошибке компиляции).

static void Main(string[] args) {

**// Передача в виде списка значений, разделенных запятыми,.**…

 double average;

 average = CalculateAverage(4.0, 3.2, 5.7);

 Console.WriteLine("Среднее 4.0, 3.2, 5.7 равно: {0}", average);

 //… **или передача в виде массива значений.**

 double[] data = {4.0, 3.2, 5.7};

 average = CalculateAverage(data);

 Console.WriteLine ("Среднее равно: {0}", average);

 Console.ReadLine();

}

Это завершает наше вводное обсуждение модификаторов параметров. Мы снова обратимся к этой теме немного позже (в этой же главе), когда будем обсуждать различия между типами значений и ссылочными типами. А пока что давайте рассмотрим итерационные и условные конструкции языка программирования C#.

**Исходный код.**Проект SimpleParams размещен в подкаталоге, соответствующем главе 3.

**Итерационные конструкции**

Все языки программирования предлагают конструкции обеспечивающие возможность повторения блоков программного кода, пока не выполнено условие завершения повторений. Если у вас есть опыт программирования, то операторы цикла в C# будут для вас понятными и не должны требовать пространных объяснений.

В C# обеспечиваются следующие четыре итерационные конструкции:

• цикл for;

• цикл foreach/in;

• цикл while;

• цикл do*/*while.

Давайте рассмотрим все указанные конструкции по очереди.

**Цикл for**

Когда требуется повторить блок программного кода определенное число раз, оператор for подходит для этого лучше всего. Вы можете указать, сколько раз должен повториться блок программного кода, а также условие окончания цикла. Без лишних объяснений, вот вам соответствующий образец синтаксиса.

**// База для цикла.**

static void Main(string[] args) {

***//*Переменная 'i' доступна только в контексте этого цикла for.**

 for(int i **=**0; i **‹**10; i++) {

  Console.WriteLine("Значение переменной: {0} ", i);

 }

**// Здесь переменная 'i' недоступна.**

}

Все ваши привычные приемы использования циклов C, C++ и Java применимы и при построении операторов for в C#. Вы можете создавать сложные условия окончания цикла, строить бесконечные циклы, а также использовать ключевые слова goto, continue и break. Я думаю, что эта итерационная конструкция будет вам понятна. Если же вам требуются дальнейшие объяснения по поводу ключевого слова fоr в C#, используйте документацию .NET Framework 2.0 SDK.

**Цикл foreach**

Ключевое слово C# foreach позволяет повторить определенные действия для всех элементов массива без необходимости выяснения размеров массива. Вот два примера использования foreach, один для массива строк, а другой – для массива целых, чисел.

**// Прохождение массива с помощью foreach.**

static void Main(string[] args) {

 string[] books = {"Сложные алгоритмы", "Классическая технология COM", "Язык C# и платформа .NET"};

 foreach(**string**s in books) Console.WriteLine(s);

 int[] myInts = {10, 20*,*30, 40};

 foreach(**int**i in myInts) Console*.*Writeline(i);

}

В дополнение к случаю простых массивов, foreach можно использовать и для просмотра системных и пользовательских коллекций. Обсуждение соответствующих подробностей предполагается в главе 7, поскольку этот вариант применения ключевого слова foreach предполагает понимание интерфейсного программирования и роли интерфейсов IEnumerator и IEnumerable.

**Конструкции while и do/while**

Цикл while оказывается полезным тогда, когда блок операторов должен выполняться до тех пор, пока не будет достигнуто заданное условие. Конечно, при этом требуется, чтобы в области видимости цикла while было определено условие окончания цикла, иначе вы получите бесконечный цикл. В следующем примере сообщение "в цикле while" будет печататься до тех пор, пока пользователь не введет значение "да" в командной строке.

static void Main(string[] args) {

 string userIsDone = "нет";

**// Проверка на соответствие строке в нижнем регистре.**

**while**(userIsDone.ToLower() != "да") {

  Console.Write("Вы удовлетворены? [да] [нет]: ");

  userIsDone = Console.ReadLine();

  Console.WriteLine{"В цикле while");

 }

}

Цикл do/while подобен циклу while. Как и цикл while, цикл do/while используется для выполнения последовательности действий неопределенное число раз. Разница в том, что цикл do/while гарантирует выполнение соответствующего блока программного кода как минимум один раз (простой цикл while может не выполниться ни разу, если условие его окончания окажется неверным с самого начала).

static void Main(string[] args) {

 string userlsDone = "";

**do** {

  Console.WriteLine("В цикле do/while");

  Console.Write("Вы удовлетворены? [да] [нет]: ");

  userIsDone = Console.ReadLine();

 } **while**(userIsDone.ToLower() != "да"); **// Обратите внимание на точку с запятой!**

}

**Конструкции выбора решений и операции сравнения**

В C# определяются две простые конструкции, позволяющие изменить поток выполнения программы по набору условий:

• оператор if/else;

• оператор switch.

**Оператор if/else**

В отличие от C и C++, оператор if/else в C# может работать только с булевыми выражениями, а не с произвольными значениями -1, 0. Поэтому в операторах if/else обычно используются операции C#, показанные в табл. 3.6. чтобы получить буквальные булевы значения.

**Таблица 3.6.**Операции сравнения в C#

|  |  |  |
| --- | --- | --- |
| **Операция сравнения** | **Пример использования** | **Описание** |
| == | if (age == 30) | Возвращает true (истина) только в том случае, когда  выражении одинаковы |
| != | if("Foo"!= myStr) | Возвращает true (истина) только в том случае, когда  выражения различны |
| ‹ › ‹= ›= | if(bonus‹2000) if(bonus›2000)  if(bonus‹=2000) if(bonus›=2000) | Возвращает true (истина) только в том случае, когда  выражение А соответственно меньше, больше, меньше  или равно, больше или равно выражению В |

Программистам, использующим C и C++, следует обратить внимание на то, что их привычные приемы по проверке условий "на равенство нулю" в C# работать не будут*.*Например, вы хотите выяснить, будет ли данная строка длиннее пустой строки. Может возникнуть искушение написать следующее.

**// В C# это недопустимо, поскольку Length возвращает int, а не bool.**

string thoughtOfThеDay = "Старую coбaку новым трюкам научить МОЖНО";

if (thoughtOfTheDay.Length) {

*…*

}

В данном случае для использования cвойства String.Length нужно изменить условие так, как показано ниже.

**// Это допустимо, так как результатом будет true или false.**

if (0 != thoughtOfTheDay.Length)

Чтобы обеспечить более сложную проверку, оператор if может содержать сложные выражения и другие операторы, Синтаксис C# в данном случае идентичен C(++) и Java (и не слишком отличается от Visual Basic). Для построения сложных выражений C# имеет вполне отвечающий ожиданиям набор условных операций, описания которых предлагаются в табл. 3.7.

**Таблица 3.7.**Условные операции в C#

|  |  |  |
| --- | --- | --- |
| **Операция** | **Пример** | **Описание** |
| && | if ((age == 30)&& (name == "Fred")) | Условная операция AND (И) |
| || | if ((age == 30) || (name == "Fred")) | Условная операция OR (ИЛИ) |
| ! | if (!myBool) | Условная операция NOT (HE) |

**Оператор switch**

Другой простой конструкцией выбора, предлагаемой в C#, является оператор switch. Как и в других языках типа C, оператор switch позволяет обработать поток выполнения программы на основе заданного набора вариантов. Например, следующий метод Main() позволяет печатать строку, зависящую от выбранного варианта (случай default предназначен для обработки непредусмотренных вариантов выбора).

// **Переключение по числовому значению.**

static void Main(string[] args) {

 Console.WriteLine("1 [C#], 2 [VB]");

 Console.Write("Выберите язык, который вы предпочитаете: ");

 string langChoice = Console.ReadLine();

 int n = int.Parse(langChoice);

 switch (n) {

 case 1:

  Console.WriteLine("Отлично! C# – это прекрасный язык.");

  break;

 case 2:

  Console.WriteLine("VB .NET: ООП, многозадачность и т.д.!");

  break;

 default:

  Console.WriteLine("Хорошо… удачи вам с таким выбором!");

  break;

 }

}

**Замечание.**В C# требуется, чтобы каждый вариант выбора (включая default), содержащий выполняемые операторы, завершался оператором break или goto, во избежание прохода сквозь структуру при невыполнении условия.

Приятной особенностью оператора switch в C# является то, что в рамках этого оператора вы можете оценивать не только числовые, но и строковые данные. Именно это используется в следующем операторе switch (при таком подходе нет необходимости переводить пользовательские данные в числовые значения).

static void Main(string[] args) {

 Console.WriteLine("C# или VB");

 Console.Write("Выберите язык, который вы предпочитаете: ");

 string langChoice = Console.ReadLine();

 switch (langChoice) {

 case "C#":

  Console.WriteLine("Отлично! C# – это прекрасный язык. ");

  break;

 case "VB":

  Console.WriteLine("VB .NET: ООП, многозадачность и т.д.!");

  break;

 default:

  Console.WriteLine("Хорошо… удачи вам с таким выбором!");

  break;

 }

}

**Исходный код*.***Проект IterationsAndDeсisions размещен в подкаталоге, соответствующем главе 3.

**Типы, характеризуемые значениями, и ссылочные типы**

Подобно любому другому языку программирования, язык C# определяет ряд ключевых слов, представляющих базовые типы данных, такие как целые числа, символьные данные, числа с плавающим десятичным разделителем и логические (булевы) значения. Если вы работали с языком C++, то будете рады узнать, что здесь эти внутренние типы являются "фиксированными константами", т.е., например, после создания элемента целочисленных данных все языки .NET будут понимать природу этого типа и диапазон его значений,

Тип данных .NET может либо характеризоваться значением, либо быть ссылочным типом (т.е. характеризоваться ссылкой). К типам, характеризуемым значением, относятся все числовые типы данных (int, float и т.д.), а также перечни и структуры, размещаемые в *стеке.*Поэтому типы, характеризуемые значениями, можно сразу же удалить из памяти, как только они оказываются вне контекста их определений.

**// Целочисленные данные характеризуются значением!**

public void SomeMethod() {

 int i = 0;

 Console.WriteLine(i);

} **// здесь *'*i*'*удаляется из стека.**

Когда вы присваиваете один характеризуемый значением тип другому, по умолчанию выполняется "почленное" копирование. Для числовых и булевых типов данных единственным "членом", подлежащим копированию, является непосредственное значение переменной,

**// Для типов, характеризуемых значениями, в результате такого**

**// присваивания в стек помещаются две независимые переменные.**

public void SomeMethod() {

 int i = 99;

 int j = i;

**// После следующего присваивания значением 'i' останется 99.**

 j = 8732;

}

Этот пример может не содержать для вас ничего нового, но важно понять, что .NET-структуры (как и перечни, которые будут рассмотрены в этой главе позже) тоже являются типами, характеризуемыми значением. Структуры, в частности, дают возможность использовать основные преимущества объектно-ориентированного подхода (инкапсуляции) при сохранении эффективности размещения данных в стеке. Подобно классам, структуры могут использовать конструкторы (с аргументами) и определять любое число членов.

Все структуры неявно получаются из класса System.ValueType. С точки зрения функциональности, единственной целью System.ValueType является "переопределение" виртуальных методов System.Object (этот объект будет описан чуть позже) с целью учета особенностей семантики типов, заданных значениями, в противоположность ссылочным типам. Методы экземпляра, определенные с помощью System.ValueType, будут идентичны соответствующим методам System.Object.

**// Структуры и перечни являются расширениями System.ValueType.**

public abstract class ValueType: object {

 public virtual bool Equals(object obj);

 public virtual int GetHashCode();

 public Type GetType();

 public virtual string ToString();

}

Предположим, что вы создали C#-структуру с именем MyPoint, используя ключевое слово C# struct.

**// Структуры являются типами, которые характеризуются значениями.**

struct MyPoint {

 public int x, у;

}

Чтобы разместить в памяти тип структуры, можно использовать ключевое слово new, что, кажется, противоречит интуиции, поскольку обычно подразумевается, что new всегда размещает данные в динамически распределяемой памяти. Это частица общего "тумана", сопровождающего CLR. Мы можем полагать, что вообще все в программе является объектами и значениями, создаваемыми с помощью new. Однако в том случае, когда среда выполнения обнаруживает тип. полученный из System.ValueType, выполняется обращение к стеку.

**// Все равно используется стек!**

MyPoint р = new MyPoint();

Структуры могут создаваться и без использования ключевою слова new.

MyPoint p1;

p1.x = 100;

p1.y = 100;

Но при использовании этого подхода вы *должны*выполнить инициализацию всех полей данных до их использования. Если этого не сделать, возникнет ошибка компиляции.

**Типы, характеризуемые значениями, ссылочные типы и оператор присваивания**

Теперь изучите следующий метод Main() и рассмотрите его вывод, показанный на рис. 3.12.

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Типы, характеризуемые значением / Ссылочные типы \*\*\*");

 Console.WriteLine(''-› Создание p1");

 MyPoint p1 = new MyPoint();

 p1.x = 100;

 p1.у = 100;

 Console.WriteLine("-› Приcваивание p1 типу p2\n");

 MyPoint p2 = p1;

 // Это p1.

 Console.WriteLine"p1.x = {0}*",*p1.x);

 Console.WriteLine"p1.y = {0}*",*p1.y);

 // Это р2.

 Console.WriteLine("p2.x = {0}", p2.x);

 Console.WriteLine("p2.у = {0}", p2.y);

 // Изменение p2.x. Это НЕ влияет на p1.x.

 Console.WriteLine("-› Замена значения p2.x на 900");

 р2.х = 900;

 // Новая печать.

 Console.WriteLine("-› Это снова значения х… ");

 Console.WriteLine("p1.x = {0}", p1.x);

 Console.WriteLine("p2.x = {0}", р2.х);

 Console ReadLine();

}

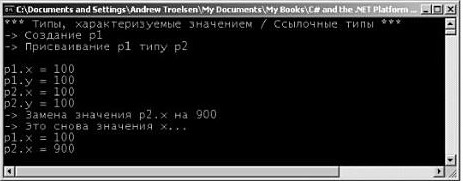


Рис. 3.12. Для типов, характеризуемых значениями, присваивание означает буквальное копирование каждого поля

Здесь создается переменная типа MyPoint (с именем p1), которая затем присваивается другой переменной типа MyPoint (р2). Ввиду того, что MyPoint является типом, характеризуемым значением, в результате в стеке будет две копии типа MyPoint, каждая из которых может обрабатываться независимо одна от другой. Поэтому, когда изменяется значение р2.х, значение p1.x остается прежним (точно так же, как в предыдущем примере с целочисленными данными).

Ссылочные типы (классы], наоборот, размещаются в управляемой динамически распределяемой памяти (managed heap). Эти объекты остаются в памяти до тех пор, пока сборщик мусора .NET не уничтожит их. По умолчанию в результате присваивания ссылочных типов создается новая ссылка на *тот же*объект в динамической памяти. Для иллюстрации давайте изменим определение типа MyPoint со структуры на класс.

**// Классы всегда оказываются ссылочными типами,**

**class** MyPoint { **// ‹= Теперь это класс!**

 public int х, у;

}

Если выполнить программу теперь, то можно заметить изменения в ее поведении (рис. 3.13).

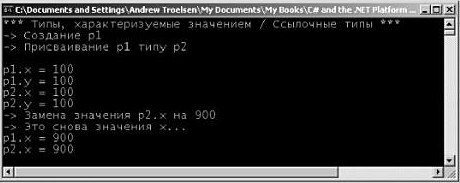


Рис. 3.13. Для ссылочных типов присваивание означает копирование ссылки

В данном случае имеется две ссылки на один и тот же объект в управляемой динамической памяти. Поэтому, если изменить значение x с помощью ссылки р2, то мы увидим, что p1.х укажет на измененное значение.

**Типы, характеризуемые значениями и содержащие ссылочные типы**

Теперь, когда вы чувствуете разницу между типами, характеризуемыми значением, и ссылочными типами, давайте рассмотрим более сложный пример. Предположим, что имеется следующий ссылочный тип (класс), обрабатывающий информационную строку, которую можно установить с помощью пользовательского конструктора.

class ShapeInfo {

 public string infoString;

 public ShapeInfo(string info) { infoString = info; }

}

Предположим также, что вы хотите поместить переменную этого типа класса в тип с именем MyReсtangle (прямоугольник), характеризуемый значением. Чтобы позволить внешним объектам устанавливать значение внутреннего поля ShapeInfо, вы должны создать новый конструктор (при этом конструктор структуры, заданный по умолчанию, является зарезервированным и не допускает переопределения).

struct MyRectangle {

 // **Структура MyRectangle содержит член ссылочного типа.**

 public ShapeInfo reсtInfo;

 public int top, left, bottom, right;

 public MyRactangle(string info) {

  rectInfo = new ShapeInfo(info);

  top = left = 10;

  bottom = right = 100;

 }

}

Теперь вы имеете ссылочный тип. внутри типа, характеризуемого значением. И здесь возникает вопрос на миллион долларов: что случится, если присвоить одну переменную типа MyRectangle другой такой же переменной? С учетом того, что вы уже знаете о типах, характеризуемых значениями, вы можете сделать правильное предположение о том, что целые данные (которые на самом деле и формируют эту структуру) для каждой переменной MyRectangle должны быть независимыми элементами. Но что можно сказать о внутреннем ссылочном типе? Будет скопировано полное состояние этого объекта или будет скопирована *ссылка* на этот объект? Проанализируйте следующий программный код и рассмотрите рис. 3.14, который может подсказать правильный ответ.

static void Main(string[] args) {

 // **Создание первого объекта MyRectangle.**

 Console.WriteLine("-› Создание r1");

 MyRectangle r1 = new MyRectangle("Это мой первый прямоугольник");

 // **Присваивание новому MyRectangle значений r1**.

 Console.WriteLine("-› Присваивание r1 типу r2");

 MyRectangle r2;

 r2 = r1;

 // **Изменение значений r2.**

 Console.WriteLine("-› Изменение значений r2");

 r2.rectInfo.InfoString = "Это новая информация!");

 r2.bottom = 4444;

 // **Print values**

 Console.WriteLine("-› Значения после изменений:");

 Console.WriteLine("-› r1.rectInfo.infoString: {0}", r1.rectInfo.infoString);

 Console.WriteLine("-› r2.rectInfo.infoString: {0}", r2.rectInfo.infoString);

 Console.WriteLine("-› r1.bottom: {0}", r1.bottom);

 Console.WriteLine("-› r2.bottom: {0}", r2.bottom);

}

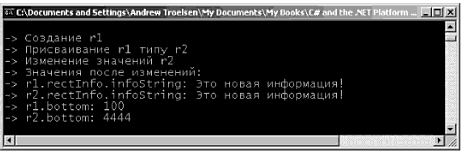


Рис. 3.14. Внутренние ссылки указывают на один и тот же объект

Как видите, при изменении значения информирующей строки с помощью ссылки r2 ссылка r1 отображает точно такое же значение. По умолчанию, когда тип, характеризуемый значением, содержит ссылочные типы, присваивание приводит к копированию *ссылок.*В результате вы получаете две независимые структуры, каждая из которых содержит ссылки, указывающие на один и тот же объект в памяти (т.е. "поверхностную копию"). Если вы хотите иметь "детальную копию", когда состояние внутренних ссылок полностью Копируется в новый объект, необходимо реализовать интерфейс ICloneable (это будет обсуждаться в главе 7).

**Исходный код.**Проект ValAndRef размещен в подкаталоге, соответствующем главе 3.

**Передача ссылочных типов по значению**

Очевидно, что ссылочные типы могут передаваться членам типов, как параметры. Но передача объекта по ссылке отличается от его передачи по значению. Чтобы понять суть различий, предположим, что у нас есть класс Person (персона), определенный следующим образом.

class Person {

 public string fullName;

 public byte age;

 public Person(string n, byte a) {

  fullName = n;

  age = a;

 }

 public Person() {}

 public void PrintInfo() { Console.WriteLine("{0}, {1} года (лет)", fullName, age); }

}

Теперь создадим метод, который позволяет вызывающей стороне переслать тип Person по значению (обратите внимание на отcутcтвие модификаторов параметра).

public static void SendAPersonByValue(Person p) {

 // **Изменяет ли это возраст 'р'?**

 p.age = 92;

**// Увидит ли вызывающая сторона такие изменения?**

 p*=*new Person ("Никки", 192);

}

Обратите внимание на то, что метод SendAPersonByValue() пытается изменить получаемую ссылку Person на новый объект, а также изменить некоторые данные состояния. Давайте проверим работу этого метода, используя следующий метод Main().

static void Main(string[] args) {

 // **Передача ссылочных типов по значению.**

 Console.WriteLine("\*\*\* Передача объекта Person по значению \*\*\*");

 Person fred = new Persоn("Фред", 2);

 Console.WriteLine("Person до вызова по значению");

 fred.PrintInfo()*;*

 SendAPersonByValue(fred);

 Console.WriteLine("Persоn после вызова по значению");

 fred.PrintInfо();

}

На рис. 3.15 показан соответствующий вывод.

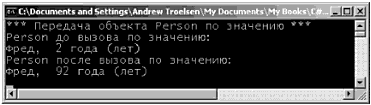


Рис. 3.15. Передача ссылочных типов по значению блокирует соответствующую ссылку

Как видите, значение возраста (age) изменяется. Кажется, такое поведение при передаче параметра противоречит самому термину "по значению". Если вы способны изменить состояние получаемого объекта Person, что же все-таки копируется? Ответ здесь следующий: в объект вызывающей стороны копируется *ссылка.*Поэтому, поскольку метод SendAPersonByValue() и объект вызывающей стороны указывают на один и тот же объект, можно изменить состояние данных объекта. Что здесь *невозможно,*так это изменить саму ссылку так, чтобы она указывала на другой объект (это напоминает ситуацию с постоянными указателями в C++).

**Передача ссылочных типов по ссылке**

Теперь предположим, что у нас есть метод SendAPersonByReference(), который передает ссылочный тип по ссылке (обратите внимание на то, что здесь присутствует модификатор параметра ref).

public static void SendAPersonByReference(**ref** Person p) {

**// Изменение некоторых данных 'р'.**

 p.age = 122;

**// Теперь 'р' указывает на новый объект в динамической памяти!**

 р = new Person("Никки", 222);

}

Как вы можете догадаться сами, это обеспечивает вызывающей стороне полную гибкость в управлении входными параметрами. Вызывающая сторона не только может изменить состояние объекта, но и переопределить ссылку так, чтобы она указывала на новый тип Person. Рассмотрите следующий вариант.

static void Main(string[] args) {

**// Передача ссылочных типов по ссылке.**

 Console.WriteLine("\n\*\*\* Передача объекта Person по ссылке \*\*\*");

 Person mel = new Person("Мэл", 23);

 Console.WriteLine("Person до вызова по ссылке:");

 mel.PrintInfo();

 SendAPersonByReference(ref mel);

 Console.WriteLine("Person после вызова по ссылке:");

 mel.PrintInfо();

}

Из рис. 3.16 видно, что тип с именем Мэл возвращается после вызова как тип с именем Никки.



Рис. 3.16. Передача ссылочных типов по ссылке позволяет перенаправить ссылку

Золотым правилом при передаче ссылочных типов по ссылке является следующее.

• Если ссылочный тип передается по ссылке, то вызывающая сторона может изменить не только состояние данных соответствующего объекта, но *сам объект ссылки.*

**Исходный код.** Проект RefTypeValTypeParams размещен в подкаталоге, соответствующем главе 3.

**Типы, характеризуемые значениями, и ссылочные типы: заключительные замечания**

Чтобы завершить обсуждение данной темы, изучите информацию табл. 3.8, в которой приводится краткая сводка основных отличий между типами, характеризуемыми значением, и ссылочными типами.

Таблица 3.8. Сравнение типов, характеризуемых значением, и ссылочных типов

|  |  |  |
| --- | --- | --- |
| **Вопрос** | **Тип, характеризуемый значением** | **Ссылочный тип** |
| Где размещается тип? | В стеке | В управляемой динамический памяти |
| Как представляется переменная? | В виде локальной копии | В виде ссылки на место в памяти, занятое соответствующим экземпляром |
| Что является базовым типом? | Оказывается производным от System.ValueType | Может получаться из любого типа, (кроме System.ValueType), не являющегося изолированным (подробности в главе 4) |
| Может ли тип быть базовым для других типов? | Нет. Типы, характеризуемые значениями, всегда  изолированы и не могут быть расширены | Да. Если тип не изолирован, он может быть базовым для других типов |
| Каким является поведение, принятое по умолчанию при передаче параметров? | Переменные передаются по значению (т.е. вызванной  функции передается копия переменной) | Переменные передаются по ссылке (например, в вызванную функцию передается адрес переменной) |
| Может ли тип переопределить System.Object.Finalize()? | Нет. Типы, характеризуемые значениями, никогда не  размещаются в динамической памяти и поэтому не  требуют финализации | Да, неявно (подробности в главе 4) |
| Можно ли определить конструкторы для этого типа? | Да, но конструктор, заданный по умолчанию,  является зарезервированным (т.е., другие  конструкторы обязательно должны иметь аргументы) | Безусловно! |
| Когда переменные данного типа прекращают свое существование? | Когда они оказываются вне контекста определения | Когда для управляемой динамической памяти выполняется сборка мусора |

Несмотря на указанные отличия, и типы, характеризуемые значением, и ссылочные типы могут реализовывать интерфейсы и поддерживать любое число полей, методов, перегруженных операций, констант, свойств и событий.

**Операции создания объектного образа и восстановления из объектного образа**

Ввиду того, что в .NET определяются две главные категории типов (характеризуемые значением или ссылкой), может понадобиться представление переменной одной категории в виде переменной другой категории. В C# предлагается очень простой механизм, называемый *операцией создания объектного образа*(boxing), позволяющий превратить тип, характеризуемый значением, в ссылочный тип. Предположим, что вы создали переменную типа short.

// **Создание значения типа short.**

short s =25;

Если в приложении потребуется конвертировать этот тип значения в ссылочный тип, вы должны "упаковать" это значение так, как показано ниже.

// **"Упаковка" значения в объектную ссылку.**

**object**objShort = s;

Операцию создания объектного образа можно формально определить, как процесс явного преобразования типа, характеризуемого значением, в соответствующий ссылочный тип с помощью сохранения переменной в System.Object. Когда значение преобразуется в объектный тип, среда CLR размещает новый объект в динамической памяти и копирует значение соответствующего типа (в данном случае это значение 25) в созданный экземпляр. Вам возвращается ссылка на новый размещенный в памяти объект. При использований такого подхода у разработчика .NET не возникает необходимости использовать интерфейсные классы, чтобы временно обращаться с данными стека как с объектами, размещенными в динамической памяти.

Обратная операция тоже предусмотрена, и называется она *восстановлением из объектного образа*(unboxing). Восстановление из объектного образа является процессом обратного преобразования значения, содержащегося в объектной ссылке, в значение соответствующего типа, размещаемое в стеке. Операция восстановления из объектного образа начинается с проверки того, что тип данных, в который выполняется восстановление, эквивалентен типу, который был приведён к объекту. Если это так, то выполняется обратное копирование соответствующего значения в локальную переменную в стеке. Например, следующая операция восстановления из объектного образа будет выполнена успешно, поскольку соответствующий тип objShort действительно имеет тип short (операцию преобразования типов в C# мы рассмотрим подробно в следующей главе, а пока что не слишком беспокойтесь о деталях).

**// Обратное преобразование ссылки в соответствующее значение short.**

short anotherShort = (short)objShort;

Снова обратим ваше внимание на то, что восстановление следует выполнять в соответствующий тип данных. Так, следующая программная логика восстановления из объектного образа генерирует исключение InvalidCastException (обсуждение вопросов обработки исключений содержится в главе 6).

**// Некорректное восстановление из объектного образа.**

static void Main(string[] args) {

 …

 try {

**// Тип в "yпаковке" - это HE int, a shоrt!**

  int i = (int)objShort;

 }  catch(InvalidCastExceptien e) {

  Console.WriteLine("ОЙ!\n{0} ", e.ToString());

 }

}

**Примеры создания объектных образов и восстановления значений**

Вы, наверное, спросите, когда действительно бывает необходимо вручную выполнять преобразование в объектный тип (или восстановление из объектного образа)? Предыдущий пример был исключительно иллюстративным, поскольку в нем для данных short не было никакой реальной необходимости приведении к объектному типу (с последующим восстановлением данных из объектного образа).

Реальность такова, что необходимость вручную приводить данные к объектному типу возникает очень редко – если возникает вообще. В большинстве случаев компилятор C# выполняет такие преобразования автоматически. Например, при передаче типа, характеризуемого значением, методу, предполагающему получение объектного параметра, автоматически "в фоновом режиме" происходит приведение к объектному типу.

class Program {

 static void Main(string[] args) {

 // **Создание значения int (тип, характеризуемый значением).**

 int myInt = 99;

 // **myInt передается методу, предполагающему**

**// получение объекта, поэтому myInt приводится**

**// к объектному типу автоматически.**

 UseThisObject(myInt);

 Console.ReadLine();

}

static void UseThisObject(**object**o) {

 Console.WriteLine("Значением о является: {0}", о);}

}

Автоматическое преобразование в объектный тип происходит и при работе c типами библиотек базовых классов .NET. Например, пространство имен System.Collections (формально оно будет обсуждаться в главе 7) определяет тип класса с именем ArrayList. Подобно большинству других типов коллекций, ArrayList имеет члены, позволяющие вставлять, получать и удалять элементы.

public class System.Collections.ArrayList: object, System.Collections.IList, System.Collections.ICollection, System.Collections.IEnumerable, ICloneable {

 …

 public virtual int Add**(object**value);

 public virtual void Insert(int index, **object**value);

 public virtual void Remove**(object**obj);

 public virtual **object**this[int index] {get; set;}

}

Как видите, эти члены действуют на типы System.Object. Поскольку все, в конечном счете, получается из этого общего базового класса, следующий программный код оказывается вполне корректным.

static void Main(string [] args) {

 …

 ArrayList myInts = new ArrayList();

 myInts.Add(88);

 myInts.Add(3.33);

 myInts.Add(false);

}

Но теперь с учетом вашего понимания ссылочных типов и типов, характеризуемые значением, вы можете спросить: что же на самом деле размещается в ArrayList? (Ссылки? Копии ссылок? Копии структур?) Как и в случае, с рассмотренным выше методом UseThisObject(), должно быть ясно, что каждый из типов данных System.Int32 перед размещением в ArrayList в действительности приводится к объектному типу. Чтобы восстановить элемент из типа ArrayList, требуется выполнить соответствующую операцию восстановления.

static void BoxAndUnboxInts() {

 // **"Упаковка" данных int в ArrayList**.

 ArrayList myInts = new ArrayList();

 myInts.Add(88);

 myInts.Add(3.33);

 myInts.Add(false);

 // **Извлечение первого элемента из ArrayList.**

 int firstItem = (int)myInts[0];

 Console.WriteLine("Первым элементом является {0}", firstItem);

}

Без сомнения, операции приведения к объектному типу и восстановления из объектного образа требуют времени и, если эти операций использовать без ограничений, это может влиять на производительность приложения. Но с использованием такого подхода .NET вы можете симметрично работать с типами характеризуемыми значением, и со ссылочными типами.

**Замечание**. В C# 2.0 потери производительности из-за приведения к ссылочному типу и восстановления из объектного образа можно нивелировать путем использования *обобщений*(generics), которые будут рассмотрены в главе 10.

**Восстановление из объектного образа для пользовательских типов**

Когда методу, предполагающему получение экземпляров System.Object, передаются пользовательские структуры или перечни, тоже происходит их приведение к объектному типу, Однако после получения входного параметра вызванным методом вы не сможете получить доступ к каким бы то ни было членам структуры (или перечня), пока не выполните операцию восстановлений из объектного образа для данного типа. Вспомним структуру МуРoint, определенную в этой главе выше.

Struct MyPoint {

 public int x, у;

}

Предположим, что вы посылаете переменную MyPoint новому методу с именем UseBoxedMyPoint().

static void Main(string[] args) {

 …

 MyPoint p;

 p.x = 10;

 p.y = 20;

 UseBoxedMyPoint(p);

}

При попытке получить доступ к полю данных MyPoint возникнет ошибка компиляции, поскольку метод предполагает, что вы действуете на строго типизованный System.Object.

static void UseBoxedMyPoint(object o) {

 // **Ошибка! System.Object не имеет членов-переменных**

**// с именами 'х' и 'у'**.

 Console.WriteLine ("{0}, {1}", о.х, о.у);

}

Чтобы получить доступ к полю данных MyPoint, вы должны сначала восстановить параметр из объектного образа. Сначала можно использовать ключевое слово C# is для проверки того, что этот параметр на самом деле является переменной MyPoint. Ключевое слово is рассматривается в главе 4, здесь мы только предлагаем пример его использования.

static void UseBoxedMyPoint(object о) {

 if (о **is** MyPoint) {

  MyPoint p = (MyPoint)o;

  Console.WriteLine ("{0}, {1}", p.x, p.y);

 } else Console.WriteLine("Вы прислали не MyPoint.");

}

**Исходный код.**Проект Boxing размещен в подкаталоге, соответствующем главе 3.

**Работа с перечнями .NET**

Вдобавок к структурам в .NET имеется еще один тип из категории характеризуемых значением – это *перечни.*При создании программы часто бывает удобно создать набор символьных имен для представления некоторых числовых значений. Например, при создании системы учета оплаты труда работников предприятия вы предпочтете использовать константы Manager (менеджер), Grunt (рабочий), Contractor (подрядчик) и VP (вице-президент) вместо простых числовых значений {0, 1, 2, 3}. Именно по этой причине в C# поддерживаются пользовательские перечни. Например, вот перечень EmpType.

**// Пользовательский перечень.**

enum EmpType {

 Manager, // = 0

 Grunt, // = 1

 Contractor, // = 2

 VP // = 3

}

Перечень EmpType определяет четыре именованные константы, соответствующие конкретным числовым значениям. В C# схема нумерации по умолчанию предполагает начало с нулевого элемента (0) и нумерацию последующих элементов по правилам арифметической прогрессии n + 1. При необходимости вы имеете возможность изменить такое поведение на более удобное.

**// начало нумерации со значения 102.**

enum EmpType {

 Manager = 102,

 Grunt, // = 103

 Contractor, // =104

 VP // = 105

}

Перечни не обязаны следовать строгому последовательному порядку нумерации элементов. Если (по некоторой причине) вы сочтете разумным создать EmpType так, как показано ниже, компилятор примет это без возражений.

//**Элементы перечня не обязаны следовать в строгой последовательности!**

enum EmpType {

 Manager = 10,

 Grunt = 1,

 Contractor = 100,

 VP = 9

}

Тип, используемый для каждого элемента в перечне, по умолчанию отображается в System.Int32. Такое поведение при необходимости тоже можно изменить. Например, если вы хотите, чтобы соответствующее хранимое значение EmpTyре было byte, а не int, вы должны написать следующее.

**// Теперь EmpType отображается в byte.**

enum EmpType: **byte**{

 Manager = 30,

 Grunt = 1,

 Contractor = 100,

 VP = 9

}

**Замечание.**Перечни в C# могут определяться в унифицированной форме для любого из числовых типов (byte, sbyte, short, ushort, int, uint, long или ulong). Это может быть полезно при создании программ для устройств с малыми объемами памяти, таких как КПК или сотовые телефоны, совместимые с .NET.

Установив диапазон и тип хранения для перечня, вы можете использовать его вместо так называемых "магических чисел". Предположим, что у вас есть класс, определяющий статическую функцию с единственным параметром EmpType.

static void AskForBonus**(EmpType e)**{

 switch(e) {

 case **EmpType.Contractor:**

  Console.WriteLine("Вам заплатили достаточно…");

  break;

 case **EmpType.Grunt**:

  Console.WriteLine("Вы должны кирпичи укладывать…");

  break;

 case **EmpType.Manager**:

  Console.WriteLine("Лучше скажите, что там с опционами!");

  break;

 case **EmpType.VP:**

  Console.WriteLine("ХОРОШО, сэр!");

  break;

 default:

  break;

 }

}

Этот метод можно вызвать так.

static void Main(string[] args) {

 // **Создание типа contractor.**

 EmpType fred;

 fred = **EmpType.Contractor;**

 AskForBonus(fred);

}

**Замечание.**При ссылке на значение перечня всегда следует добавлять префикс имени перечня (например, использовать EmpType.Grunt, а не просто Grunt).

**Базовый класс System.Enum**

Особенностью перечней .NET является то, что все они неявно получаются из System.Enum. Этот базовый класс определяет ряд методов, которые позволяют опросить и трансформировать перечень. В табл. 3.9 описаны некоторые из таких методов, и все они являются статическими.

**Таблица 3.9.**Ряд статических членов System.Enum

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Format() | Преобразует значение данного типа перечня в эквивалентное строковое  представление в соответствии с указанным форматом |
| GetName() GetNames() | Возвращает имя (или массив имен) для константы с указанным значением |
| SetUnderlyingType() | Возвращает тип данных, используемый для хранения значений данного  перечня |
| GetValues() | Возвращает массив значений констант данного перечня |
| IsDefined() | Возвращает признак существования в данном перечне константы с  указанным значением |
| Parse() | Преобразует строковое представление имен или числовых значений одной  или нескольких констант перечня в эквивалентный объект перечня |

Статический метод Enum.Format() можно использовать с флагами форматирования, которые рассматривались выше при обсуждении System.Console. Например, можно извлечь строку c именем (указав G)*,*шестнадцатиричное (X) или числовое значение (D, F и т.д.).

В System.Enum также определяется статический метод GetValues(). Этот метод возвращает экземпляр System.Array (мы обсудим этот объект немного позже), в котором каждый элемент соответствует паре "имя-значение" данного перечня. Для Примера рассмотрите следующий фрагмент программного кода.

static void Main (string[] args) {

 // **Печать информации для перечня EmpType**.

 Array obj = Enum.GetValues(typeof(EmpType));

 Console.WriteLine("В этом перечне {0} членов.", obj.Length);

 foreach(EmpType e in obj) {

  Console.Write("Строка с именем: {0},", e.ToString());

  Console.Write("int: ({0}), ", Enum.**Format**(typeof(EmpType)*,*e, "D"));

  Console.Write("hex: ({0})\n", Enum.**Format**(typeof(EmpType), e, "X"));

 }

}

Как вы сами можете догадаться, этот блок программного кода для перечня EmpType печатает пары "имя-значение" (в десятичном и шестнадцатиричном формате).

Теперь исследуем свойство IsDefined. Это свойство позволяет выяснить, является ли данная строка членом данного перечня. Предположим, что нужно выяснить, является ли значение SalesPerson (продавец) частью перечня EmpType. Для этого вы должны послать указанной функции информацию о типе перечня и строку, которую требуется проверить (информацию о типе можно получить с помощью операции typeof, которая подробно рассматривается в главе 12).

static void Main(string[] args) {

 …

 // **Есть ли значение SalesPerson в EmpType?**

 if (Enum.**IsDefined(**typeof(EmpType), "SalesPerson")) Console.WriteLine("Да, у нас есть продавцы.");

 else Console.WriteLine("Нет, мы работаем без прибыли…");

}

С помощью статического метода Enum.Parse() можно генерировать значения перечня, соответствующие заданному строковому литералу. Поскольку Parse() возвращает общий System.Object, нужно преобразовать возвращаемое значение в нужный тип.

**// Печатает "Sally is a Manager"**.

EmpType sally = (EmpType)Enum.Parse(typeof(EmpType), "Manager");

Console.WriteLine("Sally is a {0}", sally.ToString());

И последнее, но не менее важное замечание: перечни в C# поддерживают различные операции, которые позволяют выполнять сравнения с заданными значениями, например:

static void Main(string[] args) {

…

 // **Какая из этих переменных EmpType**

**// имеет большее числовое значение?**

 EmpType Joe = EmpType.VP;

 EmpType Fran = EmpType.Grunt;

 if **(Joe**‹ **Fran)**Console.WriteLine("Значение Джо меньше значения Фрэн.");

 else Console.WriteLine("Значение Фрэн меньше значения Джо.");

}

**Исходный код.**Проект Enums размещен в подкаталоге, соответствующем главе 3.

**Мастер-класс: System.Object**

**Совет.**Следующий обзор System.Object предполагает, что вы знакомы с понятиями виртуального метода и переопределения методов. Если мир ООП для вас является новым, вы можете вернуться к этому разделу после изучения материала главы 4.

В .NET каждый тип в конечном итоге оказывается производным от общего базового класса System.Object. Класс Object определяет общий набор членов, поддерживаемых каждым типом во вселенной .NET. При создании класса, для которого не указан явно базовый класс, вы неявно получаете этот класс из System.Object.

**// Неявное получение класса из System.Object.**

class.HelloClass {…}

Если вы желаете уточнить свои намерения, операция C#, обозначаемая двоеточием (:), позволяет явно указать базовый класс типа (например. System.Object).

**// В обоих случаях класс явно получается из System.Object.**

class ShapeInfo: System.Object {…}

class ShapeInfo: object {…}

Тип System.Object определяет набор членов экземпляра и членов класса (статических членов). Заметим, что некоторые из членов экземпляра объявляются с использованием ключевого слова virtual и поэтому могут *переопределяться*порождаемым классом.

**// Класс, занимающий наивысшую позицию в .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();

  public static bool Equals(object objA, object objB);

  public static bool ReferenceEquals(object objA, object objB);

 }

}

В табл. 3.10 предлагаются описания функциональных возможностей методов экземпляра для указанного объекта.

**Таблица 3.10.**Наиболее важные члены System.Object

|  |  |
| --- | --- |
| **Метод экземпляра класса Object** | **Описание** |
| Equals() | По умолчанию возвращает true (истина), когда сравниваемые элементы  ссылаются на один и тот же элемент в памяти. Поэтому используется для  сравнения объектных ссылок, а не состояний объектов. Обычно  переопределяется так, чтобы значение true возвращалось тогда, когда  сравниваемые объекты имеют одинаковые значения внутреннего состояния  (те, одинаковую семантику значений). При переопределении Equals()  следует также переопределить GetHashCode() |
| GetHashCode() | Возвращает целое значение, идентифицирующее объект в памяти. Если вы  собираетесь разместить определяемые вами типы в типе System.Collections.  Hashtable, рекомендуется переопределить заданную по умолчанию  реализацию этого члена |
| GetType() | Возвращает объект System.Туре, полностью описывающий данный элемент.  Это RTTI-метод (RunTime Type Identification – идентификация типа в среде  выполнения), доступный для всех объектов (соответствующие вопросы  обсуждаются в главе 12) |
| ToString() | Возвращает строковое представление данного объекта в формате  пространствоИмен.имяТипа (т.е. полное, или абсолютное имя). Если тип  определен не в рамках пространства имен, возвращается только имяТипа.  Этот метод может переопределяться подклассом и возвращать не абсолютное  имя, а строку пар имен и значений, представляющих внутреннее состояние  объекта |
| Finalize() | Этот защищенный метод (если он переопределен) вызывается средой  выполнения .NET, когда объект удаляется из динамической памяти.  Соответствующий процесс сборки мусора рассматривается в главе 5 |
| MemberwiseClone() | Защищенный метод, возвращающий новый объект, который является  "почленной" копией данного объекта. Если объект содержит ссылки на  другие объекты, то копируются ссылки на соответствующие типы  (т.е. выполняется поверхностное копирование). Если объект содержит типы,  характеризуемые значениями, получаются полные копии значений |

**Поведение System.Object, заданное по умолчанию**

**Чтобы продемонстрировать некоторые особенности принятого по умолчанию**поведения базового класса System.Object, рассмотрим класс Person (персона), определенный в пользовательском пространстве имен ObjectMethods.

**// Ключевое слово 'namespace' обсуждается в конце этой славы.**

namespace ObjectMethods {

 class Person {

  public Person(string fname, string lname, string s, byte a) {

   firstName = fname; lastName = lname; SSN = s; age = a;

  }

  public Person() {}

**// Персональные данные (данные состояния)**.

  public string firstMame;

  public string lastName;

  public string SSN;

  public byte age;

 }

}

Теперь используем тип Person в рамках метода Main().

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Работа с классом Object \*\*\*\*\*\n");

 Person fred = new Person("Фред", "Кларк", "111-11-1111", 20);

 Console.WriteLine("-› fred.ToString: {0}", fred.ToString());

 Console.WriteLine("-› fred.GetHashCode: {0}*",*fred.GetHashCode());

 Console.WriteLine("-› базовый класс для 'fred': {0}", fred.GetType().BaseType);

 // **Создание дополнительных ссылок на 'fred'**.

 Person p2 = fred;

 object о = p2;

 // **Указывали ли все 3 экземпляра на один объект в памяти?**

 if (о.Equals(fred) && p2.EqualS(o)) Console.WriteLine("fred, p2 и о ссылаются на один объект!*"*);

 Console*.*ReadLine();

}

На риc. 3.17 показан вариант вывода, полученного при тестовом запуске программы.

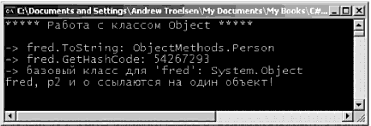


Рис. 3.17. Реализация членов System.Object, заданная по умолчанию

Обратите внимание на то, что заданная по умолчанию реализация ToString() просто возвращает полное имя типа (например, в виде пространствоИмён.ИмяТипа).

Метод GetType() возвращает объект System.Type, который определяет свойство BaseType (как вы можете догадаться сами, оно идентифицирует полное имя базового класса данного типа).

Теперь рассмотрим программный код, использующий метод Equals(). Здесь в управляемой динамической памяти размещается новый объект Person, и ссылка на этот объект запоминается в ссылочной переменной fred. Переменная р2 тоже имеет тип Person, однако здесь не создается новый экземпляр класса Person, a присваивается fred переменной р2. Таким образом, и fred, и р2, а также переменная о (типа object, которая была добавлена для полноты картины) указывают на один и тот же объект в памяти. По этой причине тест на тождественность будет успешным.

**Переопределение элементов System.Object, заданных по умолчанию**

Хотя заданное по умолчанию поведение System.Object может оказаться вполне приемлемым в большинстве случаев, вполне обычным для создаваемых вами типов будет переопределение некоторых из унаследованных методов. В главе 4 предлагается подробный анализ возможностей ООП в рамках C#, но, по сути, *переопределение*- это изменение поведения наследуемого *виртуального*члена в производном классе. Как вы только что убедились, System.Object определяет ряд виртуальных методов (например, ToString() и Equals()), задающих предусмотренную реализацию. Чтобы иметь другую реализацию этих виртуальных членов для производного типа, вы должны использовать ключевое слово C# override (букв. подменять).

**Переопределение System.Object.ToString()**

Переопределение метода ToString() дает возможность получить "снимок" текущего состояния объекта. Это может оказаться полезным в процессе отладки. Для примера давайте переопределим System.Object.ToString() так, чтобы возвращалось текстовое представление состояния объекта (обратите внимание на то, что здесь используется новое пространство имен System.Text).

**// Нужно сослаться на System.Text для доступа к StringBuilder.**

using System;

using System.Text;

class Person {

 // **Переопределение System.Object.ToString()**.

 public override string ToString() {

  StringBuilder sb = new StringBuilder();

  sb.AppendFormat("[FirstName={0}; ", this.firstName);

  sb.AppendFormat(" Lastname={0}; ", this, lastName);

  sb.AppendFormat(" SSN={0};", this.SSN);

  sb.AppendFormat(" Age={0}]", this.age);

  return sb.ToString();

 }

 …

}

To, как вы форматируете строку, возвращающуюся из System.Object.ToString(), не очень важно. В данном примере пары имен и значений помещены в квадратные скобки и разделены точками с запятой (этот формат используется в библиотеках базовых классов .NET).

В этом примере используется новый тип System.Text.StringBuilder, который будет подробно описан позже. Здесь следует только подчеркнуть, что StringBuilder обеспечивает более эффективную альтернативу конкатенации строк в C#.

**Переопределение System.Object. Equals()**

Давайте переопределим и поведение System.Object.Equals(), чтобы иметь возможность работать с *семантикой, основанной на значениях.*Напомним, что по умолчанию Equals() возвращает true (истина), когда обе сравниваемые ссылки указывают на один и тот же объект в динамической памяти. Однако часто бывает нужно не то, чтобы две ссылки указывали на один объект в памяти, а чтобы два объекта имели одинаковые состояния (в случае Person это означает равенство значений name, SSN и age).

public override bool Equals(object о) {

**// Убедимся, что вызывающая сторона посылает**

**// действительный объект Person.**

 if (о!= null && о is Person) {

**// Теперь проверим, что данный объект Person**

**// и текущий объект (this) несут одинаковую информацию.**

Person temp = (Person)о;

  if (temp.firstName == this.firstName && temp.lastName == this.lastName && temp*.*SSN == this.SSN && temp.age == this.age) return true;

 }

 return falsе; // **He одинаковую!**

}

Здесь с помощью ключевого слова is языка C# вы сначала проверяете, что вызывающая сторона действительно передает методу Equals() объект Person. После этого нужно сравнить значение поступающего параметра со значениями полей данных текущего объекта (обратите внимание на использование ключевого слова this, которое ссылается на текущий объект).

Прототип System.Object.Equals() предполагает получение единственного аргумента типа object. Поэтому вы должны выполнить явный вызов метода Equals(), чтобы получить доступ к членам типа Person. Если значения name, SSN и age двух объектов будут идентичны, вы имеете два объекта с одинаковыми данными состояния, поэтому возвратится true (истина). Если какие-то данные будут различаться, вы получите false (ложь).

Переопределив System.Object.ToString() для данного класса, вы получаете очень простую возможность переопределения System.Object.Equals(). Если возвращаемое из ToString() значение учитывает все члены текущего класса (и данные базовых классов), то метод Equals() может просто сравнить значения соответствующих строковых типов.

public override bool Equals(object o) {

 if (o != null && о is Person) {

  Person temp = (Person)o;

  if (this.ToString() == о.ToString()) return true;

  else return false;

 }

 return false;

}

Теперь предположим, что у нас есть тип Car (автомобиль), экземпляр которого мы попытаемся передать методу Person.Equals().

**// Автомобили – это не люди!**

Car с = new Car();

Person p = new Person();

p.Equals(c);

Из-за проверки в среде выполнения на "истинность" объекта Person (с помощью оператора is) метод Equals() возвратит false. Теперь рассмотрим следующий вызов.

**// Ой!**

Person р = new Person();

p.Equals(null);

Это тоже не представляет опасности, поскольку наша проверка предусматривает возможность поступления пустой ссылки.

**Переопределение System.Object.GetHashCode()**

Если класс переопределяет метод Equals(), следует переопределить и метод System.Object.GetHashCode(). Не сделав этого, вы получите предупреждение компилятора. Роль GetHashCode() – возвратить числовое значение, которое идентифицирует объект в зависимости от его состояния. И если у вас есть два объекта Person, имеющие идентичные значения name, SSN и age, то вы должны получить для них одинаковый хеш-код.

Вообще говоря, переопределение этого метода может понадобиться только тогда, когда вы собираетесь сохранить пользовательский тип в коллекции, использующей хеш-коды, например, в System.Collections.Hashtable. В фоновом режиме тип Hashtable вызывает Equals() и GetHashCode() содержащихся в нем типов, чтобы определить правильность объекта, возвращаемого вызывающей стороне. Поскольку System.Object не имеет информации о данных состояния для производных типов, вы должны переопределить GetHashCode() для всех типов, которые вы собираетесь хранить в Hashtable.

Есть много алгоритмов, которые можно использовать для создания хеш-кода, как "изощренных", так и достаточно "простых". Еще раз подчеркнем, что значение хеш-кода объекта зависит от состояния этого объекта. Класс System.String имеет довольно солидную реализацию GetHashCode(), основанную на значении символьных данных. Поэтому, если можно найти строковое поле, которое будет уникальным для всех рассматриваемых объектов (например, поле SSN для объектов Person), то можно вызвать GetHashCode() для строкового представлении такого поля.

**// Возвращает хеш-код на основе SSN.**

public **override**int GetHashCode() {

 return SSN.GetHashCode();

}

Если вы не сможете указать подходящий элемент данных, но переопределите ToString(), то можно просто возвратить хеш-код строки, возвращенной вашей реализацией ToString().

// **Возвращает хеш-код на основе пользовательского ToString**().

public override int GetHashCode() {

 return **ToString**().GetHashCode();

}

**Тестирование переопределенных членов**

Теперь можно проверить обновленный класс Person. Добавьте следующий программный код в метод Main() и сравните результат его выполнения с тем, что показано на рис. 3.18.

static void Main (string[] args) {

 // **ВНИМАНИЕ: эти объекты должны быть идентичными.**

 Person р3 = new Person("Fred", "Jones", "222-22-2222", 98);

 Person p4 = new Person("Fred", "Jones", "222-22-2222", 98);

 // **Тогда эти хеш-коды и строки будут одинаковыми.**

 Console.WriteLine("-› Хеш-код для р3 = {0}", р3.getHashCode());

 Console.WriteLine("-› Хеш-код для р4 = {0}", p4.GetHashCode());

 Console.WriteLine("-› Строка для р3 = {0}", p3.ToString());

 Console.WriteLine("-› Cтрока для р4 = {0}", p4.ToString());

**// Здесь состояния должны быть одинаковыми.**

 if (р3.Equals(p4)) Console.WriteLine("-› Состояния р3 и р4 одинаковы!");

 else Console.WriteLine("-› Состояния р3 и р4 различны!");

 // **Изменим age для р4**.

 Console.WriteLine("\n-› Изменение age для р4\n");

 р4.age = 2;

**// Теперь состояния неодинаковы: хеш-коды и строки будут разными.**

 Console.WriteLine("-› Строка для р3 = {0}", p3.ToString());

 Console.WriteLine("-› Строка для р4 = {0}", p4.ToString());

 Console.WriteLine("-› Хеш-код для р3 = {0}", р3.GetHashCode());

 Console.WriteLine("-› Хеш-код для р4 = {0}", p4.GetHashCode());

 if (р3.Equals(p4)) Console.WriteLine("-› Состояния р3 и р4 одинаковы!")

 else Console.WriteLine("-› Состояния р3 и р4 различны!");

}

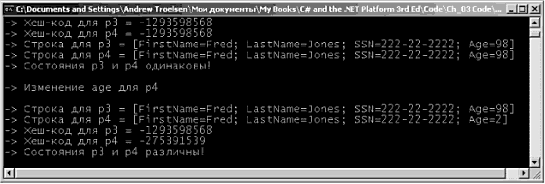


Рис. 3.18. Результаты переопределения членов System.Object

**Статические члены System.Object**

В завершение нашего обсуждения базового класса .NET, находящегося на вершине иерархии классов, следует отметить, что System.Object определяет два статических члена (Object.Equals() и Object.ReferenceEquals()), обеспечивающих проверку на равенство значений и ссылок соответственно. Рассмотрим следующий программный код.

static void Main(string[] args) {

 // **Два объекта с идентичной конфигурацией.**

 Person р3 = new Person("Fred", "Jones", "222-22-2222", 98);

 Person p4 = new Person("Fred", "Jones", "222-22-2222", 98);

 // **Одинаковы ли состояния р3 и р4? ИСТИНА!**

 Console.WriteLine("Одинаковы ли состояния: р3 и р4: {0} ", object.Equals(р3, р4));

 // **Являются ли они одним объектом в памяти? ЛОЖЬ!**

 Console.WriteLine ("Указывают ли р3 и р4 на один объект: {0} ", object.ReferenceEquals(р3, р4));

}

**Исходный код.**Проект ObjectMethods размещен в подкаталоге, соответствующем главе 3.

**Типы данных System (и их обозначения в C#)**

Вы, наверное, уже догадались, что каждый внутренний тип данных C# – это на самом деле сокращенное обозначение некоторого типа, определенного в пространстве имен System. В табл. 3.11 предлагается список типов данных System, указаны их диапазоны изменения, соответствующие им псевдонимы C# и информация о согласованности типа со спецификациями CLS.

**Таблица 3.11.**Типы System и их обозначения в C#

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Обозначение в C#** | **Согласованность с CLS** | **Тип System** | **Диапазон изменения** | **Описание** |
| sbyte | Нет | System.SByte | от -128 до 127 | 8-битовое число со знаком |
| byte | Да | System.Byte | От 0 до 255 | 8-битовое число без знака |
| short | Да | System.Int16 | от -32768 до 32767 | 16-битовое число со знаком |
| ushort | Нет | System.UInt16 | от 0 до 65535 | 16-битовое число без знака |
| int | Да | System.Int32 | от -2147483648 до  2147483647 | 32-битовое число со знаком |
| Uint | Нет | System.UInt32 | от 0 до 4294967295 | 32-битовое число без знака |
| long | Да | System.Int64 | От -9223372036854775808 до  9223372036854775807 | 64-битовое число со знаком |
| ulong | Нет | System.UInt64 | от 0 до 18446744073709551615 | 64-битовое число без знака |
| char | Да | System.Char | от U0000 до Uffff | Отдельный 16-битовый символ Unicode |
| float | Да | System.Single | от 1.5×10-45 до 3.4×1038 | 32-битовое число с плавающим десятичным разделителем |
| double | Да | System.Double | от 5.0х10-324до 1.7х10308 | 64-битовое число с плавающим десятичным разделителем |
| bool | Да | System.Boolean | true или false | Представляет истину или ложь |
| decimal | Да | System.Decimal | от 100 до 1028 | 96-битовое число со знаком |
| string | Да | System.String | Ограничено системной памятью | Представляет набор символов Unicode |
| object | Да | System.Object | Любой тип можно сохранить в  объектной переменной | Базовый класс всех типов во вселенной .NET |

**Замечание.**По умолчанию действительный числовой литерал справа от операции присваивания интерпретируется, как double. Поэтому, чтобы инициализировать переменную типа float, используйте суффикс f или F (например 5.3F).

Интересно отметить, что и примитивные типы данных .NET организованы в иерархии классов. Отношения между этими базовыми типами (как и некоторыми другими типами, с которыми мы познакомимся чуть позже) можно представить так, как показано на рис. 3.19.

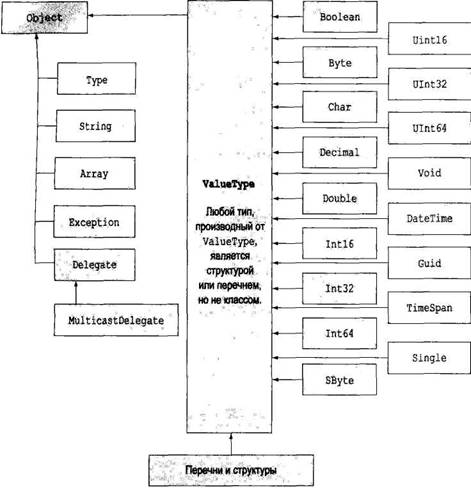


Рис. 3.19. Иерархия типов System

Как видите, каждый из этих типов, в конечном счете, получается из System.Object. Ввиду того, что такие типы данных, как, например, int являются просто сокращенными обозначениями соответствующего системного типа (в данном случае типа System.Int32), следующий вариант синтаксиса оказывается вполне допустимым.

**// Помните! В C# int - это просто сокращение для System. Int32.**

Console.WriteLine(12.GetHashCode());

Console.WriteLine(12.Equals(23));

Console.WriteLine(12.ToString());

Console.WriteLine(12); // **ToString() вызывается автоматически**.

Console.WriteLine(12.GetType().BaseType);

К тому же, поскольку все типы значений имеют конструктор, заданный по умолчанию, можно создавать системные типы с помощью ключевого слова new, в результате чего переменной, к тому же, будет присвоено значение по умолчанию. Хотя использование ключевого слова new при создании типов данных System выглядит несколько "неуклюжим", следующая конструкция оказывается в C# синтаксически правильной.

**// Следующие операторы эквивалентны.**

bool b1 = new bool(); **// b1= false.**

bool b2 = false;

Кстати, заметим, что можно создавать системные типы данных, используя абсолютные имена.

**// Следующие операторы также семантически эквивалентны.**

System.Вoоl b1 *=*new System.Bool(); **// b1 = false.**

System.Bool sb2 = false;

**Эксперименты с числовыми типами данных**

Числовые типы .NET поддерживают свойства MaxValue и МinValue, сообщающие информацию о диапазоне данных, которые может хранить данный тип. Предположим, что мы создали несколько переменных типа System.UInt16 (unsigned short – короткое целое без знака), как показано ниже.

static void Main(string[] args) {

 System.Uint16.myUInt16 = 300000;

 Console.WriteLine("Максимум для UInt16: {0} ", UInt16.**MaxValue**);

 Console.WriteLine("Минимум для UInt16: {0} ", UInt16.**MinValue**);

 Console.WriteLine("Значение равно: {0} ", **myUInt16**);

 Console.WriteLine("Я есть: {0} ", myUInt16.**GetType()**);

**// Теперь для сокращения System.UInt16 (т.e для ushort).**

 ushort myOtherUInt16 = 12000;

 Console.WriteLine("Максимум для UInt16: {0} ", ushort.**MaxValue**);

 Console.WriteLine("Минимум для UInt16: {0} ", ushort.**MinValue**);

 Console.WriteLine("Знaчение равно: {0} ", **myOtherUInt16**);

 Console.WriteLine("Я есть: {0} ", myotherUInt16.**GetType**());

 Console.ReadLine();

}

Вдобавок к свойствам MinValue/MaxValue системные типы могут определять другие полезные члены. Например, тип System.Double позволяет получить значения Epsilon и Infinity.

Console.WriteLine("*-›*double.Epsilon: {0}", double.**Epsilon**);

Console.WriteLine("-› double.РositiveInfinitу: {0} ", double.**PositiveInfinity**);

Console.WriteLine("-› double.NegativeInfinity: {0}", double.**NegativeInfinity**);

Console.WriteLine("-› double.MaxValue: {0}", double.**MaxValue**);

Console.WriteLine("-› double.MinValue: {0}", double.**MinValue**);

**Члены System.Boolean**

Теперь рассмотрим тип данных System.Boolean. В отличие от C(++), в C# единственными возможными значениями для bool являются {true | false}. В C# вы не можете назначать типу bool импровизированные значения (например, -1, 0, 1), что считается (большинством программистов) правильным нововведением. С учетом этого должно быть понятно, почему System.Boolean не поддерживает свойства MinValue/MaxValue, а поддерживает TrueString/FalseString.

**// В C# нет произвольных типов Boolean!**

bool b = 0**; // Недопустимо!**

bool b2 = -1; **// Также недопустимо!**

bool b3 = true**; // Без проблем.**

bool b4 = false**; // Без проблем.**

Console.WriteLine("-› bool.FalseString: {0}", bool**.FalseString)**;

Console.WriteLine("-› bool.TrueString: {0}", bool**.TrueString);**

**Члены System.Char**

Текстовые данные в C# представляются встроенными типами данных string и char. Все .NET-языки отображают текстовые типы в соответствующие базовые типы (System.String и System.Char). Оба эти типа в своей основе используют Unicode.

Тип System.Char обеспечивает широкие функциональные возможности, далеко выходящие за рамки простого хранения символьных данных (которые, кстати, должны помещаться в одиночные кавычки). Используя статические методы System.Char, вы можете определить, является ли данный символ цифрой, буквой, знаком пунктуации или чем-то иным. Для иллюстрации рассмотрим следующий фрагмент программного кода.

static void Main(string[] args) {

 …

**// Проверьте работу следующих операторов…**

 Console.WriteLine("-› char.IsDigit('К'): {0}",  char.**IsDigit**('К'));

 Console.WriteLine("-› char.IsDigit('9'): {0}", char.**IsDigit**('9'));

 Console.WriteLine("-› char.IsLetter('10', 1): {0}", char.**IsLetter(**"10", 1));

 Console.WriteLine("-› char.IsLetter('p'): {0}", char.**IsLetter**('p'));

 Console.WriteLine("-› char.IsWhiteSpace('Эй, там!', 3): {0}", char.**IsWhiteSpace**("Эй, там!", 3));

 Console.WriteLine("-› char.IsWhiteSpace('Эй, там!', 4): {0}", char.**IsWhiteSpace**("Эй, там!", 4));

 Console.WriteLine("-› char.IsLettetOrDigit('?'): {0}", char.**IsLetterOrDigit**('?'));

 Console.WriteLine("-› char.IsPunctuation('!'): {0}", char.**IsPunctuation**('!'));

 Console.WriteLine("-›char.IsPunctuation('›'): {0}", char**.IsPunctuation(**'›'));

 Console.WriteLine("-› char.IsPunctuation(','): {0}", char.**IsPunctuation**(','));

 …

}

Как видите, для всех этих статических членов System.Char при вызове используется следующее соглашение: следует указать либо единственный символ, либо строку с числовым индексом, который указывает местоположение проверяемого символа.

**Анализ значений строковых данных**

Типы данных .NET обеспечивают возможность генерировать переменную того типа, который задается данным текстовом эквивалентом (т.е. выполнять синтаксический анализ). Эта возможность может оказаться чрезвычайно полезной тогда, когда требуется преобразовать вводимые пользователем данные (например, выделенные в раскрывающемся списке) в числовые значения. Рассмотрите следующий фрагмент программного кода.

static void Main(string[] args) {

…

 bool myBool = bool.**Parse**("True");

 Console.WriteLine("-› Значение myBool: {0}", myBool);

 double myDbl*=*double.**Parse**("99,884");

 Console.WriteLine("-› Значение myDbl: {0}", myDbl);

 int myInt = int**.Parse**("8");

 Console.WriteLine("-› Значение myInt: {0}", myInt);

 Char myChar = char**.Раrsе**("w")**;**

 Console.WriteLine(''-› Значение myChar: {0}\n", myChar);

 …

}

**System.DateTime и System.TimeSpan**

В завершение нашего обзора базовых типов данных позволите обратить ваше внимание на то, что пространство имен System определяет несколько полезных типов данных, для которых в C# не предусмотрено ключевых слов. Это, в частности, типы DateTime и TimeSpan (задачу исследования типов System.Guid и System.Void, которые среди прочих показаны на рис. 3.19, мы оставляем на усмотрение заинтересованных читателей).

Тип DateTime содержит данные, представляющие конкретные дату (месяц, день, год) и время, которые можно отформатировать различными способами с помощью соответствующих членов. В качестве простого примера рассмотрите следующий набор операторов.

static void Main (string[] args) {

 …

**// Этот конструктор использует (год, месяц, день)**

 DateTime dt = new DateTime(2004, 10, 17);

 // **Какой это день недели?**

 Console.WriteLine("День {0} – это (1}", dt.Date, dt.DayOfWeek);

 dt.AddMonths(2); // **Теперь это декабрь.**

 Console.WriteLine ("Учет летнего времени: {0}", dt.IsDaylightSavingTime());

 …

}

Структура TimeSpan позволяет с легкостью определять и преобразовывать единицы времени с помощью различных ее членов, например:

static void Main(string[] args) {

 …

 // **Этот конструктор использует (часы, минуты, секунды)**

 TimeSpan ts = new TimeSpan(4, 30, 0);

 Console.WriteLine(ts);

 // **Вычтем 15 минут из текущего значения TimeSpan и распечатаем результат.**

 Console.WriteLine(ts.Subtract(new TimeSpan (0, 15, 0)));

 …

}

На рис. 3.20 показан вывод операторов DateTime и TimeSpan.

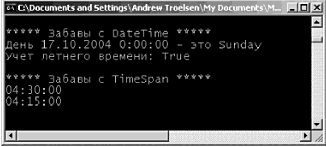


Рис. 3.20. Использование типов DateTime и TimeSpan

**Исходный код.**Проект DataTypes размещен в подкаталоге, соответствующем главе 3.

**Тип данных System.String**

Ключевое слово string в C# является сокращенным обозначением типа System.String, предлагающего ряд членов, вполне ожидаемых от этого класса. В табл. 3.12 предлагаются описания некоторых (но, конечно же, не всех) таких членов.

Таблица 3.12. Некоторые члены System.String

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Length | Свойство, возвращающее длину текущей строки |
| Contains() | Метод, применяемый для выяснения того, содержит ли текущий строковый объект  данную строку |
| Format() | Статический метод, применяемый для форматировании строковых литералов с  использованием примитивов (числовых данных и других строк) и обозначений  типа {0}, уже встречавшихся ранее в этой главе |
| Insert() | Метод, используемый для получения копии текущей строки, содержащей  добавляемые строковые данные |
| PadLeft() PadRight() | Методы, возвращающие копии текущей строки, дополненные указанными  данными в качестве заполнителя |
| Remove() Replace() | Методы, используемые для получения копии строки с соответствующими  модификациями (при удалении или замене символов) |
| Substring() | Метод, возвращающий строку, которая представляет подстроку текущей  строки |
| ToCharArray() | Метод, возвращающий массив символов, из которых состоит текущая  строка |
| ToUpper() ToLower() | Методы, создающие копию данной строки, представленную  символами в верхнем или, соответственно, нижнем регистре |

**Базовые операции со строками**

Для иллюстрации некоторых базовых операций со строками рассмотрим следующий метод Main().

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы со строками \*\*\*\*\*");

 string s = "Boy, this is taking a long time.";

 Console.WriteLine("-› Содержит ли s 'oy'?: {0}", s**.Contains**("oy"));

 Console.WriteLine("-› Содержит ли s 'Boy'?: {0}", s.**Contains**("Boy"));

 Console.WriteLine(s**.Replace**('.', '!'));

 Console.WriteLine.(s**.Insert**(0, "Boy O' "));

 Console.ReadLine();

}

Здесь мы создаем тип string, вызывающий методы Contains(), Replace() и Insert(). Cоответствующий вывод показан на рис. 3.21.

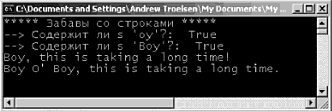


Рис. 3.21. Базовые операции во строками

Вы должны учесть то, что хотя string и является ссылочным типом, операции равенства и неравенства (== и !=) предполагают сравнение *значений*со строковыми объектами, а не областей памяти, на которые они ссылаются. Поэтому следующее сравнение в результате дает true:

string s1 = "Hello";

string s2 = "Hello";

Console.WriteLine("s1 == s2: {0}", s1 == s2);

тогда как следующее сравнение возвратит false:

string s1 = "Hello";

string s2 = "World!";

Console.WriteLine("s1 == s2: {0}", s1 == s2);

Для конкатенации существующих строк в новую строку, которая является объединением исходных, в C# предлагается операция +, как статический метод String.Concat(). С учетом этого следующие операторы оказываются функционально эквивалентными.

**// Конкатенация строк.**

string newString = s + s1 + s2;

Console.WriteLine ("s + s1 + s2 = {0}", newString);

Console.WriteLine("string.Concat(s, s1, s2) = {0}", string.**Concat**(s, s1, s2));

Другой полезной возможностью, присущей типу string, является возможность выполнения цикла по всем отдельным символам строки с использованием синтаксиса, аналогичного синтаксису массивов. Формально говоря, объекты, поддерживающие доступ к своему содержимому, подобный по форме доступу к массивам, используют *метод индексатора.*О том, как строить индексаторы, вы узнаете из главы 9, но здесь для иллюстрации соответствующего понятия предлагается рассмотреть следующий фрагмент программного кода, в котором каждый символ строкового объекта s1 выводится на консоль.

**// System.String определяет индексатор для доступа**

**// каждому символу в строке.**

for (int k = 0; k ‹ s1.Length; k++) Console.WriteLine("Char {0} is {1}", k, s1[k]);

В качестве альтернативы взаимодействию с индексатором типа можно использовать строковый класс в конструкции foreach. Ввиду того, что System.String поддерживает массив индивидуальных типов System.Char, следующий программный тоже выводит каждый символ si на консоль.

foreach (char c in s1) Console.WriteLine(с);

**Управляющие последовательности**

Как и в других языках, подобных C, строковые литералы в C# могут содержать различные управляющие последовательности, которые интерпретируются как определенный набор данных, предназначенных для отправки в выходной поток. Каждая управляющая последовательность начинается с обратной косой черты, за которой следует интерпретируемый знак. На тот случай, если вы подзабыли значения управляющих последовательностей, в табл. 3.13 предлагаются описания тех из них, которые используются чаще всего.

**Таблица 3.13.**Управляющие последовательности строковых литералов

|  |  |
| --- | --- |
| **Управляющая последовательность** | **Описание** |
| \' | Вставляет в строковый литерал знак одиночной кавычки |
| \" | Вставляет в строковый литерал знак двойной кавычки |
| \\ | Вставляет в строковый литерал знак обратной косой черты, Это может  оказаться полезным при указании пути |
| \а | Инициирует системный звуковой сигнал (beep). Для консольных приложений  это может быть аудиоподсказкой пользователю |
| \n | Вставляет знак перехода на новую строку (на платформах Win32). |
| \r | Вставляет знак возврата каретки |
| \t | Вставляет в строковый литерал знак горизонтальной табуляции |

Так, чтобы напечатать строку, в которой между любыми двумя словами имеется знак табуляции, можно использовать управляющую последовательность \t.

// **Строковые литералы могут содержать любое число**

**// управляющих последовательностей**.

string s3 = "Эй, \tвы,\tтам,\tопять!";

Console.WriteLine(s3);

Для другого примера предположим, что вам нужно создать строковый литерал, который содержит кавычки, литерал, указывающий путь к каталогу, и, наконец, литерал, который вставляет три пустые строки после вывода всех символьных данных. Чтобы не допустить появления сообщений об ошибках компиляции, используйте управляющие символы \", \\ и \n.

Consolе.WriteLine("Все любят \"Hello World\");

Console. WriteLine("C:\\MyApp\\bin\\debug");

Console.WriteLine("Все завершено.\n\n\n");

**Буквальное воспроизведение строк в C#**

В C# вводится использование префикса @ для строк, которые требуется *воспроизвести буквально.* Используя буквальное воспроизведение строк, вы отключаете обработку управляющих символов строк. Это может быть полезным при работе со строками, представляющими каталоги и сетевые пути. Тогда вместо использования управляющих символов \\ можно использовать следующее.

**// Следующая строка должна воспроизводиться буквально,**

**// поэтому все 'управляющее символы' будут отображены.**

Console.WriteLine(@"C:\MyАрр\bin\debug");

Отметьте также и то, что буквально воспроизводимые строки могут использоваться для представления пропусков пространства в строковых значениях, "растянутых" на несколько строк.

**// В буквально воспроизводимых строках**

**// пропуски пространства сохраняются.**

string myLongString = @"Это очень

      очень

          очень

              длинная строка";

Console.WriteLine(myLongString);

Двойную кавычку в такой строковый литерал можно вставить с помощью дублирования знака ", например:

Console.WriteLine(@"Cerebus said ""Darrr! Pret-ty sun-sets""");

**Роль System.Text.StringBuilder**

Тип string прекрасно подходит для того, чтобы представлять базовые строковые переменные (имя, SSN и т.п.), но этого может оказаться недостаточно, если вы создаете программу, в которой активно используются текстовые данные. Причина кроется в одной очень важной особенности строк в .NET: значение строки после ее определения изменить нельзя. Строки в C# неизменяемы.

На первый взгляд, это кажется невероятным, поскольку мы привыкли присваивать новые значения строковым переменным. Однако, если проанализировать методы System.String, вы заметите, что методы, которые, *как кажется,*внутренне изменяют строку, на самом деле возвращают измененную *копию*оригинальной строки. Например, при вызове ToUpper() для строкового объекта вы не изменяете буфер существующего строкового объекта, а получаете новый строковый объект в форме символов верхнего регистра.

static void Main(string[] args) {

 …

**// Думаете, что изменяете strFixed? А вот и нет!**

 System.String strFixed = "Так я начинал свою жизнь";

 Console.WriteLine(strFixed);

 string upperVersion = strFixed.ToUpper();

 Console.WriteLine(strFixed);

 Console.WriteLine("{0}\n\n", upperVersion);

 …

}

Подобным образом, присваивая существующему строковому объекту новое значение, вы фактически размещаете в процессе *новую*строку (оригинальный строковый объект в конечном итоге будет удален сборщиком мусора). Аналогичные действия выполняются и при конкатенации строк.

Чтобы уменьшить число копирований строк, в пространстве имен System.Text определяется класс StringBuilder (он уже упоминался нами выше при рассмотрении System.Object). В отличие от System.String, тип StringBuilder обеспечивает прямой доступ к буферу строки. Подобно System.String, тип StringBuilder предлагает множество членов, позволяющих добавлять, форматировать, вставлять и удалять данные (подробности вы найдете в документации .NET Framework 2.0 SDK).

При создании объекта StringBuilder можно указать (через аргумент конструктора) начальное число символов, которое может содержать объект. Если этого не сделать, то будет использоваться "стандартная емкость" StringBuilder, по умолчанию равная 16. Но в любом случае, если вы увеличите StringBuilder больше заданного числа символов, то размеры буфера будут переопределены динамически.

Вот пример использования этого типа класса.

using System;

using System.Text; **// Здесь 'живет' StringBuilder.**

class StringApp {

 static void Main(string[] args) {

  StringBuilder myBuffer = new StringBuilder("Моя строка");

  Console.WriteLine("Емкость этого StringBuilder: {0}", myBuffer.Capacity);

  myBuffer.**Append**(" содержит также числа:");

  myBuffer.**AppendFormat**("{0}, {1}.", 44, 99);

  Console.WriteLine("Емкость этого StringBuilder: {0}", myBuffer.Сарасitу);

  Console.WriteLine(myBuffer);

 }

}

Во многих случаях наиболее подходящим для вас текстовым объектом будет System.String. Для большинства приложений потери, связанные с возвращением измененных копий символьных данных, будут незначительными. Однако при построении приложений, интенсивно использующих текстовые данные (например, текстовых процессоров), вы, скорее всего, обнаружите, что использование System.Text.StringBuilder повышает производительность.

**Исходный код.**Проект Strings размешен в подкаталоге, соответствующем главе 3.

**Типы массивов .NET**

Формально говоря, *массив*- это коллекция указателей на данные *одного и того же*вполне определенного типа, доступ к которым осуществляется по числовому индексу. Массивы являются ссылочными типами и получаются из общего базового класса System.Array. По умолчанию для .NET-мaccивов начальный индекс равен нулю, но с помощью статического метода System.Array.CreateInstance() для любого массива можно задать любую нижнюю границу для его индексов.

Массивы в C# можно объявлять по-разному. Во-первых, если вы хотите создать массив, значения которого будут определены позже (возможно после ввода соответствующих данных пользователем), то, используя квадратные скобки ([]), укажите размеры массива во время его создания. Например:

**// Создание массива строк, содержащего 3 элемента (0-2)**

string[] booksOnCOM;

booksOnCOM = new string[3];

// **Инициализация 100-элементного массива с нумерацией (0**- **99)**

string[] booksOnDotNet = new string[100];

Объявив массив, вы можете использовать синтаксис индексатора, чтобы присвоить значения его элементам.

// **Создание, заполнение и печать массива из трех строк.**

string[] booksOnCOM; booksOnCOM = new string[3];

booksOnCOM[0] = "Developer's Workshop to COM and ATL 3.0";

booksOnCOM[1] = "Inside COM";

booksOnCOM[2] = "Inside ATL";

foreach (string s in booksOnCOM) Console.WriteLine(s);

Если значения массива во время его объявления известны, вы можете использовать "сокращенный" вариант объявления массива, просто указав эти значения в фигурных скобках. Указывать размер массива в этом случае не обязательно (он вычисляется динамически), как и при использовании ключевого слова new. Так, следующие варианты объявления массива эквивалентны.

// **'Краткий' вариант объявления массива**

**// (значения во время объявления должны быть известны).**

int[] n = new int[] {20, 22, 23, 0};

int[] n3 = {20, 22, 23, 0};

И наконец, еще один вариант создания типа массива.

int[] n2 = new int[4] {20, 22, 23, 0}; // **4 элемента, {0**- **3}**

В данном случае указанное числовое значение задает *число элементов*в массиве, а не граничное сверху значение для индексов. При несоответствии между объявленным размером и числом инициализируемых элементов вы получите сообщение об ошибке компиляции.

Независимо от того, как вы объявите массив, элементам в .NET-массиве автоматически будут присвоены значения, предусмотренные по умолчанию, сохраняющиеся до тех пор, пока вы укажете иные значения. Так, в случае массива числовых типов, каждому его элементу присваивается значение 0 (или 0.0 в случае чисел с плавающим разделителем), объектам присваивается null (пустое значение), а типам Boolean – значение false (ложь).

**Массивы в качестве параметров (и возвращаемых значений)**

После создания массива вы можете передавать его, как параметр, или получать его в виде возвращаемого значения. Например, следующий метод PrintArray() получает входной массив строк и выводит каждый элемент на консоль, а метод GetStringArray() "наполняет" массив значениями и возвращает его вызывающей стороне.

static void PrintArray**(int[] myInts)**{

 for (int i = 0; i ‹ myInts.Length; i++) Console.WriteLine("Элемент {0} равен {1}", i, myInts[i]);

}

static **string[]**GetStringArray() {

 string theStrings = { "Привет", "от", "GetStringArray"};

 return theStrings;

}

Эти методы можно вызвать из метода Main(), как показано ниже.

static void Main(string[] args) {

 int[] ages={20, 22, 23, 0};

**PrintArray**(ages);

 string[] strs = **GetStringArray**();

 foreach(string s in strs) Console.WriteLine(s);

 Console.ReadLine();

}

**Работа с многомерными массивами**

Вдобавок к одномерным массивам, которые мы рассматривали до сих пор, в C# поддерживаются два варианта многомерных массивов. Первый из них – это *прямоугольный массив,* т.е. многомерный массив, в котором каждая строка оказывается одной и той же длины. Чтобы объявить и заполнить многомерный прямоугольный массив, действуйте так, как показано ниже.

static void Main(string[] args) {

 …

 // **Прямоугольный массив MD**.

 int[,] myMatrix;

 myMatrix = new int[6,6];

**// Заполнение массива (6 \* 6).**

 for (int i = 0; i ‹ 6; i++) for (int j = 0; j ‹ 6; j++) myMatrix[i, j] = i \* j;

**// Печать массива (6 \* 6).**

 for (int i = 0; i *‹*6; i++) {

  for(int j = 0; j ‹ 6; j++) Console.Write(myMatrix[i, j] + "\t");

  Console.WriteLine();

 }

 …

}

На рис. 3.22 показан соответствующий вывод (обратите внимание на прямоугольный вид массива).



Рис. 3.22. Многомерный массив

Второй тип многомерных массивов – это *невыровненный массив.*Как следует из самого названия, такой массив содержит некоторый набор массивов, каждый из которых может иметь свой верхний предел для индексов. Например:

static void Main(string[] args) {

 …

**// Невыровненный массив MD (т.е. массив массивов).**

**// Здесь мы имеем массив из 5 разных массивов.**

 int[][] myJagArray = new int[5][];

**// Создание невыровненного массива.**

 for (int i = 0; i ‹ myJagArray.Length; i++) myJagArray[i] = new int[i +7];

**// Печать каждой строки (не забывайте о том, что**

**// по умолчанию все элементы будут равны нулю!)**

 for (int i = 0; i ‹ 5; i++) {

  Console.Write("Длина строки {0} равна {1}:\t", i, myJagArray[i].Length);

  for (int j = 0; j ‹ myJagArray[i].Length; j++) Console.Write(myJagArray[i][j] + " ");

  Console.WriteLine();

 }

}

На рис. 3.23 показан соответствующий вывод (обратите, что здесь массив имеет "неровный край").

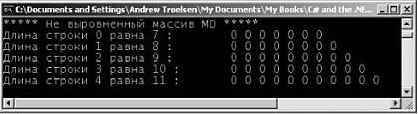


Рис. 3.23. Невыровненный массив

Теперь когда вы знаете, как строить и заполнять массивы в C#, обратим внимание на базовый класс любого массива: System.Array.

**Базовый класс System.Array**

Каждый создаваемый вами массив в .NET автоматически получается из System.Array. Этот класс определяет рад полезных методов для упрощения работы с массивами. В табл. 3.14 предлагаются описания некоторых из наиболее интересных членов указанного класса.

Таблица 3.14. Некоторые члены System.Array

|  |  |
| --- | --- |
| **Член** | **Описание** |
| BinarySearch() | Статический метод, выполняющий поиск заданного элемента в (предварительно  отсортированном) массиве. Если массив скомпонован из пользовательских типов,  искомый тип должен реализовывать интерфейс IComparer (см. главу 7),  чтобы задействовать двоичный поиск |
| Clear() | Статический метод, очищающий заданный диапазон элементов в массиве  (устанавливается 0 для числовых типов и null – для ссылочных типов) |
| CopyTo() | Метод, используемый для копирования элементов из массива-источника  в целевой массив |
| Length | Свойство, доступное только для чтения и используемое для выяснения  числа элементов в массиве |
| Rank | Свойство, возвращающее значение размерности данного массива |
| Reverse() | Статический метод, инвертирующий порядок следования элементов  одномерного массива |
| Sort() | Метод, сортирующий одномерный массив внутренних типов. Если элементы в  массиве реализуют интерфейс IComparer, можно также сортировать  пользовательские типы (снова см. главу 7) |

Рассмотрим примеры использовании некоторых из этих членов. В следующем программном коде используются статические методы Reverse() и Clear() (а также свойство Length) для вывода некоторой информации о массиве строк firstNames на консоль.

**// Создание строковых массивов и проверка**

**// некоторых членов System.Array.**

static void Main(string[] args) {

 // **Массив строк.**

 string[] firstNames = {"Steve", "Dominic", "Swallow", "Baldy"};

 // **Печать имен в объявленном виде.**

 Console.WriteLine("Вот вам массив:");

 for(int i = 0; i ‹ firstNames.Length; i++) Console.Write("Имя: {0}\t", firstNames[i]);

 Console.WriteLine("\n");

 // **Инвертирование порядка в массиве и печать.**

 Array*.*Reverse(firstNames);

 Console.WriteLine("Вот вам инвертированный массив:");

 for (int i = 0; i ‹ firstNames.Length; i++) Console.Write("Имя: (0}\t", firstNames[i]);

 Console.WriteLine("\n");

 // **Очистка всех данных, хроме Baldy.**

 Console.WriteLine("Очистка всех данных, кроме Baldy…");

 Array.Clear(firstNames, 1, 3);

 for (int i = 0; i ‹ firstNames.Length; i++) Console.Write ("Имя: {0}\t", firstNames[i]);

 Console.ReadLine();

}

Обратите особое внимание на то, что при вызове метода Clear() для массива оставшиеся элементы массива не сжимаются в меньший массив. Для подвергшихся очистке элементов просто устанавливаются значения по умолчанию. Если вам нужен контейнер динамического типа, поищите подходящий тип в пространстве имен System.Collections.

**Исходный код.**Проект Arrays размещен в подкаталоге, соответствующем главе 3.

**Типы с разрешением принимать значение null**

Вы уже видели, что типы данных CLR имеют фиксированный диапазон изменения. Например, тип данных System.Boolean может принимать значения из множества {true, false}. В .NET 2.0 можно создавать*типы с разрешением принимать значение null*(типы nullable). Тип с разрешением принимать значение null может представлять любое значение, допустимое для данного типа, и, кроме того, значение null. Так, если объявить тип System.Boolean с разрешением принимать значение null, то такой тип сможет принимать значения из множества {true, false, null}. Очень важно понимать, что тип, характеризуемый значением, без разрешения принимать значение null это значение принимать не может.

static void Main(string[] args) {

 // **Ошибка компиляции!**

**// Типы, характеризуемые значением, не допускают значений null!**

 bool myBool = null;

 int myInt = null;

}

Чтобы определить переменную типа nullable, к обозначению типа данных в виде суффикса добавляется знак вопроса (?). Такой синтаксис оказывается допустимым только тогда, когда речь идет о типах, характеризуемых значениями, или массивах таких типов. При попытке создать ссылочный тип (включая строковый) с разрешением значения null вы получите ошибку компиляции. Подобно переменным без разрешения принимать значение null, локальным переменным с таким разрешением тоже должны присваиваться начальные значения.

static void Main(string [] args) {

**// Несколько определений локальных типов**

**// с разрешенными значениями null**.

 int? nullableInt = 10;

 double? nullableDouble = 3.14;

 bool? nullableBool = null;

 char? nullableChar = 'a';

 int?[] arrayOfNullableInts = new int?[10];

 // **Ошибка! Строки являются ссылочными типами!**

 string? s = "ой!";

}

Суффикс ? в C# является сокращенной записью для указания создать переменную структуры обобщённого типа System.Nullable‹T›. Мы рассмотрим обобщений в главе 10, а сейчас важно понять то, что тип System.Nullablе‹Т› предлагает ряд членов, которые могут использовать все типы с разрешением значения null. Например, используя свойство HasValue или операцию !=, вы можете программным путем выяснить, содержит ли соответствующая переменная значение null. Значение, присвоенное типу с разрешением значения null, можно получить непосредственно или с помощью свойства Value.

**Работа с типами, для которых допустимы значения null**

Типы с разрешением принимать значение null могут оказаться исключительно полезными при взаимодействии с базами данных, где столбцы в таблице могут оказаться пустыми (т.е., неопределенными). Для примера рассмотрим следующий класс, моделирующий доступ к базе данных с таблицей, два столбца которой могут оставаться неопределенными. Обратите внимание на то, что здесь метод GetIntFromDatabase() не присваивает значение члену-переменной целочисленного типа с разрешенным значением null, в то время как GetBoolFromDatabase() назначает подходящее значение члену bool?.

Class DatabaseReader {

 // **Поле данных с разрешением значения null**.

 public int? numbericValue;

 public bool? boolValue = true;

 // **Обратите внимание на разрешение null для возвращаемого типа.**

 public int? GetIntFromDatabase() {return numberiсVаlue;}

 // **Обратите внимание на разрешение null для возвращаемого типа.**

 public bool? GetBoolFromDatabase() {return boolValue;}

}

Теперь рассмотрим следующий метод Main(), вызывающий члены класса DatabaseReader и демонстрирующий присвоенные им значения с помощью HasValue и Value в соответствии с синтаксисом C#.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с разрешением null \*\*\*\*\*\n")

 DatabaseReader dr = new DatabaseReader();

 // **Получение int из 'базы данных'**.

 int? i = dr.GetIntFromDatabase();

 if (i.HasValue) Console.WriteLine("Значение 'i' равно: {0}", i);

 else Console.WriteLine("Значение 'i' не определено.");

 // **Получение bool из 'базы данных'**.

 bool? b = dr.GetBoolFromDatabase();

 if (b != null) Console.WriteLine("Значение 'b' равно: {0}", b);

 else Console.WriteLine("Значение 'b' не определено.");

 Console.ReadLine();

}

**Операция ??**

Еще одной особенностью типов с разрешением принимать значения null, о которой вам следует знать, является то, что с такими типами можно использовать появившуюся в C# 2005 специальную операцию, обозначаемую знаком ??. Эта операция позволяет присвоить типу значение, если его текущим значением оказывается null. Для примера предположим, что в том случае, когда значение, возвращенное методом GetIntFromDatabase(), оказывается равным null, соответствующему локальному типу int с разрешением значения null нужно присвоить числовое значение 100 (конечно, упомянутый метод *всегда*возвращает null, но я думаю, вы поймете идею, которую иллюстрирует данный пример).

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с разрешением null \*\*\*\*\*\n");

 DatabaseReader dr = new DatabaseReader();

**// Если GetIntFromDatabase() возвращает null,**

**// то локальной переменной присваивается значение 100.**

 int? myData = dr.GetIntFromDatabase() ?? 100;

 Console.WriteLine("Значение myData: {0}", myData);

 Console.ReadLine();

}

**Исходный код.**Проект NullableType размещен в подкаталоге, соответствующем главе 3.

**Пользовательские пространства имен**

До этого момента мы создавали небольшие тестовые программы, используя пространства имен, существующие в среде .NET (в частности, пространство имен System). Но иногда при создании приложения бывает удобно объединить связанные типы в одном пользовательском пространстве имён. В C# это делается с помощью ключевого слова namespace.

Предположим, что вы создаете коллекцию геометрических классов с названиями Square (квадрат), Circle (круг) и Hexagon (шестиугольник). Учитывая их родство, вы хотите сгруппировать их в общее пространство имен. Здесь вам предлагаются два основных подхода. С одной стороны, можно определить все классы в одном файле (shapeslib.cs), как показано ниже.

// **shapeslib.cs**

using System;

namespace MyShapes {

 // **Класс Circle.**

 class Circle {/\* Интересные методы… \*/}

**// Класс Hexagon.**

 class Hexagon {/\* Более интересные методы… \*/}

 // **Класс Square.**

 class Square {/\* Еще более интересные методы… \*/}

}

Заметим, что пространство имен MyShapes играет роль абстрактного "контейнера" указанных типов. Альтернативным вариантом является размещение единого пространства имен в нескольких C#-файлах. Для этого достаточно "завернуть" определения различных классов в одно пространство имен.

**// circle.cs**

using System;

namespace **MyShapes**{

**// Клаcc Circle.**

 class Circle{ }

}

**// hexagon.cs**

using System;

namespace **MyShapes** {

**// Класс Hexagon.**

 class Hexagon{}

}

**// square.cs**

using System;

namespace **MyShapes**{

**// Класс Square.**

 class Square{}

}

Вы уже знаете, что в том случае, когда одному пространству имен требуются объекты из другого пространства имен, следует использовать ключевое слово using.

**// Использование типов из пространства имен MyShapes**.

using System;

using MyShapes;

namespace MyApp {

 class ShapeTester {

  static void Main(string[] args) {

   Hexagon h = new Hexagon();

   Circle с = new Circle();

   Square s = new Square();

  }

 }

}

**Абсолютные имена типов**

Строго говоря, при объявлении типа, определенного во внешнем пространстве имен, в C# не обязательно использовать ключевое слово using. Можно использовать полное, или абсолютное имя типа, которое, как следует из главы 1, состоит из имени типа с добавленным префиксом пространства имен, определяющего данный тип.

**// Заметьте, что здесь не используется 'using MyShapes'.**

using System;

namespace MyApp {

 class ShapeTester {

  static void Main(string[] args) {

   MyShapes.Hexagon h = new MyShapes.Hexagon();

   MyShapes.Circle с = new MyShapes.Circle();

   MyShapes.Square s = new MyShapes.Square();

  }

 }

}

Обычно нет необходимости использовать полное имя. Это только увеличивает объем ввода с клавиатуры, но не дает никаких преимуществ ни с точки зрения размеров программного кода, ни с точки зрения производительности программы. На самом деле в программном коде CIL типы всегда указываются с полным именем. С этой точки зрения ключевое слово using в C# просто экономит время при наборе программного кода.

Однако абсолютные имена могут быть весьма полезны (а иногда и необходимы) тогда, когда возникают конфликты при использовании пространств имен, содержащих типы с одинаковыми именами. Предположим, что есть еще одно пространство имен My3DShapes, в котором определяются три класса для визуализации пространственных форм.

**// Другое пространство форм…**

using System;

namespace My3DShapes {

 // **Трехмерный класс Circle.**

 class circle{}

**// Трехмерный класс Hexagon**.

 class Hexagon{}

 // **Трехмерный класс Square.**

 class Square{}

}

Если обновить ShapeTester, как показано ниже, вы получите целый ряд сообщений об ошибках компиляции, поскольку два пространства имен определяют типы с одинаковыми названиями.

**// Множество неоднозначностей!**

using System;

using MyShapes;

using My3DShapes;

namespace MyApp {

 class ShapeTester {

  static void Main(string[] args) {

**// На какое пространство имен ссылаются?**

   Hexagon b = new Hexagon(); // **Ошибка компиляции!**

   Circle с = new Circle(); // **Ошибка компиляции!**

   Square s = new Square(); /**/ Ошибка компиляции!**

  }

 }

}

Неоднозначность разрешится, если использовать абсолютное имя типа

**// Теперь неоднозначность ликвидирована.**

static void Main(string[] args) {

 Му3DShapes.Hexagon h = new My3DShapes.Hexagon();

 My3DShapes.Circle с = new My3DShrapes.Circle();

 MyShapes.Square s*=*new MyShapes.Square();

}

**Использование псевдонимов**

Ключевое слово C# using можно также использовать для создания псевдонимов абсолютных имен типов. После создания псевдонима вы получаете возможность использовать его как ярлык, который будет заменен полным именем типа во время компиляции. Например:

using System;

using MyShapes;

using My3DShapes;

**// Ликвидация неоднозначности с помощью псевдонима.**

using The3DHexagon = My3DShapes.Hexagon;

namespace MyApp {

 class ShapeTester {

  static void Main(string[] args) {

**// На самом деле здесь создается тип My3DShapes.Hexagon**.

   The3DHexagon h2 = new The3DHexagon();

   …

  }

 }

}

Этот альтернативный синтаксис using можно использовать и при создании псевдонимов для длинных названий пространств имен.

Одним из длинных названий в библиотеке базовых классов является System.Runtime.Serialization.Formatters.Binary. Это пространство имен содержит член с именем BinaryFormatter. Используя синтаксис using, экземпляр BinaryFormatter можно создать так, как показано ниже:

using **MyAlias**= System.Runtime.Serialization.Formatters.Binary;

namespace MyApp {

 class ShapeTester {

  static void Main(string[] args) {

**MyAlias**.BinaryFormatter b = new **MyAlias.**BinaryFormatter();

  }

 }

}

или же с помощью традиционного варианта использования директивы using.

using System.Runtime.Serialization.Formatters.Binary;

namespace MyApp {

 class ShapeTester {

  static void Main(string [] args) {

   BinaryFormatter b*=*new BinaryFormatter();

  }

 }

}

**Замечание.**Теперь в C# предлагается и механизм разрешения конфликтов для одинаково названных *пространств имен,*основанный на использовании спецификатора псевдонима пространства имен (::) и "глобальной" метки. К счастью, указанный тип коллизий возникает исключительно редко. Если вам требуется дополнительная информация по этой теме, прочитайте мою статью "Working with the C# 2.0 Command Line Compiler" (Работа с компилятором командной строки C# 2.0), которую можно найти на страницах [http://msdn.microsoft.com](http://msdn.microsoft.com/).

**Вложенные пространства имен**

Совершенствуя организацию своих типов, вы имеете возможность определить пространства имен в рамках других пространств имен. Библиотеки базовых классов .NET часто используют такие вложений, чтобы обеспечить более высокий уровень организации типа. Например, пространство имен Collections вложено в System, чтобы в результате получилась System.Collections. Чтобы создать корневое пространство имен, содержащее уже существующее пространство имен My3DShapes, можно изменить наш программный код так, как показано ниже.

**// Вложение пространства имен.**

namespace Chapter3 {

 namespace My3DShapes {

  // **Трехмерный класс Circle.**

  class Circle{}

  // **Трехмерный класс Hexagon.**

  class Hexagon{}

  // **Трехмерный класс Square.**

  class Square{}

 }

}

Во многих случаях единственной задачей корневого пространства имен является расширение области видимости, поэтому непосредственно в рамках такого пространства имен может вообще не определяться никаких типов (как в случае пространства имей Chapter3). В таких случаях вложенное пространство имен можно определить, используя следующую компактную форму.

**// Вложение пространства имен (вариант 2).**

namespace **Chapters.My3DShapes**{

**// Трехмерный класс Circle**.

 class Circle{}

 // **Трехмерный класс Hexagon.**

 class Hexagon{}

 // **Трехмерный класс Square.**

 class Square{}

}

С учетом того, что теперь пространство имен My3DShapes вложено в рамки корневого пространства имен Chapter3, вы должны изменить вид всех соответствующих операторов, использующих директиву using и псевдонимы типов.

using Chapter3.My3DShapes;

using The3DHexagon = Chapter3.My3DShapes.Hexagon;

**Пространство имен по умолчанию в Visual Studio 2005**

В заключение нашего обсуждения пространств имен следует отметить, что при создании нового C#-проекта в Visual Studio 2005 имя пространства имен вашего приложения по умолчанию будет совпадать с именем проекта. При вставке новых элементов с помощью меню Project→Add New Item создаваемые типы будут автоматически помещаться в пространство имен, используемое по умолчанию. Если вы хотите изменить имя пространства имен, используемого по умолчанию (например, так, чтобы оно соответствовало названию вашей компании), используйте опцию Default namespace (Пространство имен по умолчанию) на вкладке Application (Приложения) окна свойств проекта (рис. 3.24).

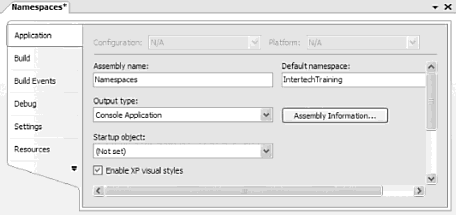


Рис. 3.24. Изменение пространства имен, используемого по умолчанию

После такой модификации любой новый элемент, вставляемый в проект, будет помещаться в пространство имен IntertechTraining (и, очевидно, чтобы использовать соответствующие типы в рамках другого пространства имен, потребуется указать подходящую по форме директиву using).

**Исходный код.**Проект Namespaces размещен в подкаталоге, соответствующем главе 3.

**Резюме**

В этой (довольно длинной) главе обсуждались самые разные аспекты языка программирования C# и платформы .NET. В центре внимания были конструкция, которые наиболее часто используются в приложениях и которые могут понадобиться вам при создании таких приложений.

Вы могли убедиться, что все внутренние типы данных в C# соответствуют определенным типам из пространства имён System. Каждый такой "системный" тип предлагает набор членов, с помощью которых программными средствами можно выяснить диапазон изменения типа. Были также рассмотрены особенности построения типов класса в C#, различные правила передачи параметров, изучены типы, характеризуемые значениями, и ссылочные типы, а также выяснена роль могущественного System.Object.

Кроме того, в главе обсуждались возможности среды CLR, позволяющие использовать объектно-ориентированный подход с общими программными конструкциями, такими как массивы, строки, структуры и перечни. Здесь же были рассмотрены операции приведения к объектному типу и восстановления из объектного образа. Этот простой механизм позволяет с легкостью переходить от типов, характеризуемых значениями, к ссылочным типам и обратно. Наконец, была раскрыта роль типов данных с разрешением принимать значение null и показано, как строить пользовательские пространства имён.

**ГЛАВА 4. Язык C# 2.0 и объектно-ориентированный подход**

В предыдущей главе мы рассмотрели ряд базовых конструкций языка C# и платформы .NET, а также некоторые типы из пространства имен System. Здесь мы углубимся в детали процесса построения объектов. Сначала мы рассмотрим знаменитые принципы ООП, а затем выясним, как именно реализуются, инкапсуляция, наследование и полиморфизм в C#. Это обеспечит знания, необходимые для того, чтобы строить иерархии пользовательских классов.

В ходе обсуждения мы рассмотрим некоторые новые конструкции, такие как свойства типов, члены типов для контроля версий, изолированные классы и синтаксис XML-кода документации. Представленная здесь информация будет базой для освоения более сложных приемов построения классов (с использованием, например, перегруженных операций, событий и пользовательских программ преобразования), рассматриваемых в следующих главах.

Даже если вы чувствуете себя вполне комфортно в рамках объектно-ориентированного подхода с другими языками программирования, я настойчиво рекомендую вам внимательно рассмотреть примеры программного кода, предлагаемые в этой главе. Вы убедитесь, что C# предлагает новые решения для многих привычных конструкций объектно-ориентированного подхода.

**Тип класса в C#**

Если вы имеете опыт создания объектов в рамках какого-то другого языка программирования, то, несомненно, знаете о роли определений классов. Формально говоря, класс – это определенный пользователем тип (User-Defined Type - UDT), который скомпонован из полей данных (иногда называемых *членами-переменными)*и функций (часто вызываемых *методами*), воздействующих на эти данные. Множество полей данных в совокупности представляет "состояние" экземпляра класса.

Мощь объектно-ориентированных языков заключается в том, что с помощью группировки данных и функциональных возможностей в едином пользовательском типе можно строить свои собственные программные типы по образу и подобию лучших образцов, созданных профессионалами. Предположим, что вы должны создать программный объект, моделирующий типичного работника для бухгалтерской программы. С минимумом требований вы можете создать класс Employee (работ-ник), поддерживающий поля для имени, текущего уровня зарплаты и ID (числового кода) работника. Дополнительно этот класс может определить метод GiveBonus(), который на некоторую величину увеличивает выплату для данного индивидуума, a также метод DisplayStats()*,*который печатает данные состояния. На рис. 4.1 показана структура типа Employee.



Pис. 4.1. Тип класса Employee

Вспомните из главы 3 что классы в C# могут определять любое число*конструкторов*- это специальные методы класса, обеспечивающие пользователю объекта простую возможность создания экземпляров данного класса с заданным поведением. Каждый класс в C# изначально обеспечивается *конструктором, заданным по умолчанию,*который по определению никогда не имеет аргументов. В дополнение к конструктору, заданному по умолчанию, можно определить любое число пользовательских конструкторов*.*

Для начала мы определим вашу первую модификацию класса Employee (по мере изучения материала данной главы мы будем добавлять в этот класс новые функциональные возможности).

**// Исходное определение класса Employee.**

namespace Employees {

 public class Employee {

  // **Поля данных.**

  private string fullName;

  private int empID;

  private float currPay;

  // **Конструкторы.**

  public Employee(){}

  public Employee(string fullName, int empID, float currPay) {

   this.fullName = fullName;

   this.empIP = empID;

   this.currPay = currPay;

  }

**// Увеличение выплаты для данного работника.**

  public void GiveBonus(float amount) { currPay *+=*amount; }

**// Текущее состояние объекта.**

  public void DisplayStats() {

   Console.WriteLine("Имя: {0} ", fullName);

   Console.WriteLine("З/п: {0} ", currPay);

   Console*.*WriteLine("Код: {0} ", empID)*;*

  }

 }

}

Обратите внимание на реализацию конструктора по умолчанию (он оказывается пустым) для класса Employee.

public class Employee {

 …

 public Employee(){}

 …

}

Подобно C++ и Java, если в определении C#-класса задаются пользовательские конструкторы, то конструктор, заданный по умолчанию, *отключается без предупреждений.*Если вы хотите позволить пользователю объекта создавать экземпляры вашего класса следующим образом:

static void Main(string[] args) {

**// Вызов конструктора, заданного до умолчанию.**

 Employee e = new Employee();

}

то должны явно переопределить конструктор, заданный по умолчанию. Если этого не сделать, то при создании экземпляра вашего класса с помощью конструктора по умолчанию вы получите ошибку компиляции. Так или иначе, следующий метод Main() создает целый ряд объектов Employee, используя наш пользовательский конструктор с тремся аргументами.

**// Создание нескольких объектов Employee.**

static void Main(string[] args) {

 Employee e = new Employee("Джо", 80, 30000);

 Employee e2;

 e2 = new Employee("Бет", 81, 50000);

 Console.ReadLine();

}

**Перегрузка методов**

Подобно другим объектно-ориентированным языкам, язык C# позволяет типу *перегружать*его методы. Говоря простыми словами, когда класс имеет несколько членов с одинаковыми именами, отличающихся только числом (или типом) параметров, соответствующий член называют *перегруженным.*В классе Employee перегруженным является конструктор класса, поскольку предложены два определения, которые отличаются только наборами параметров.

public class Employee {

 ...

 // Перегруженные конструкторы.

 public Employee(){}

 public Employee(string fullName, int empID, float currPay) {…}

 ...

}

Конструкторы, однако, не являются единственными членами, допускающими перегрузку. Продолжив рассмотрение текущего примера, предположим, что у нас есть класс Triangle (треугольник), который поддерживает перегруженный метод Draw(). С его помощью пользователю объекта позволяется выполнить визуализацию изображений, используя различные входные параметры.

public class Triangle {

 // **Перегруженный метод Draw()**.

 public void Draw(int x, int y, int height, int width) {…}

 public void Draw(float x, float y, float height, float width) {…}

 public void Draw(Point upperLeft, Point bottomRight) {…}

 public void Draw(Rect r) {…}

}

Если бы в C# не поддерживалась перегрузка методов, вы были бы вынуждены создать четыре члена с уникальными именами, что, как можете убедиться, весьма далеко от идеала.

public class Triangle {

**// Глупость**…

 public void DrawWithInts(int x, int y, int height, int width) {…}

 public void DrawWithFloats(float x, float y, float height, float width) {…}

 public void DrawWithPoints(Point upperLeft, Point bottomRight) {…}

 public void DrawWithRect(Rect r) {…}

}

Но не забывайте о том, что при перегрузке члена возвращаемый тип не может быть независимым. Так, следующий вариант просто недопустим.

public class Triangle {

 …

**// Ошибка! Нельзя перегружать методы**

**// на основе возвращаемых значений!**

 public float GetX(){…}

 public int GetX(){…}

}

**Использование this для возвратных ссылок в C#**

Обратите внимание на то, что другой конструктор класса Employee использует ключевое слово C# this.

// **Явное использование "this" для разрешения конфликтов имен.**

publiс Employee(string fullName, int empID, float currPay) {

 // **Присваивание входных параметров данным состояния.**

**this.**fullName = fullName;

**this.**empID = empID;

**this**.currPay*=*currPay;

}

Это ключевое слово C# используется тогда, когда требуется явно сослаться на поля и члены *текущего объекта.*Причиной использования ключевого слова this в этом пользовательском конструкторе является стремление избежать конфликта имен параметров и внутренних переменных состояния. Альтернативой могло бы быть изменение имен всех параметров.

**// В отсутствие конфликта имен "this" подразумевается.**

public Employee(string name, int id, float pay) {

 fullName = name;

 empID = id;

 currPay = pay;

}

В данном случае нет необходимости явно добавлять префикс this к именам членов-переменных Employee, потому что конфликт имен уже исключен. Компилятор может самостоятельно выяснить области видимости используемых членов-переменных, и в этой ситуации this называют *неявным*: если класс ссылается на свои собственные поля данных и члены-переменные (без каких бы то ни было неоднозначностей), то this подразумевается. Таким образом, предыдущая логика конструктора функционально идентична следующей.

public Employee(string name, int Id, float pay) {

**this.**fullName = name;

**this**.empID = id;

**this**.currPay = pay;

}

**Замечание.**Статические члены типа не могут использовать ключевое слово this в контексте метода. В этом есть смысл, поскольку статические члены-функции действуют на уровне класса (а не объекта). На уровне класса нет this!

**Передача вызовов конструктора с помощью this**

Другим вариантом использования ключевого слова this является такая реализация вызова одним конструктором другого, при которой не возникает избыточной логики инициализации члена. Рассмотрим следующую модификацию класса Employee.

public class Employee {

 …

 public Employee(string fullName, int empID, float currPay) {

  this.fullName = fullName;

  this.empID = empID;

  this.currPay = currPay;

 }

**// Если пользователь вызовет этот конструктор, то**

**// передать вызов версии с тремя аргументами.**

 public Employee(string fullName) : **this**(fullName, IDGenerator.GetNewEmpID(), 0.0F) {}

 …

}

Эта итерация класса Employee определяет два пользовательских конструктора, и второй из них имеет единственный параметр (имя индивидуума). Однако для построения полноценного нового Employee вы хотите гарантировать наличие соответствующего ID и значения зарплаты. Предположим, что у вас есть пользовательский класс (IDGenerator) со статическим методом GetNewEmpID(), тем или иным образом генерирующим ID нового работника. Собрав множество начальных параметров, вы передаете запрос создания объекта конструктору с тремя аргументами.

Если не передавать вызов, то придется добавить в каждый конструктор избыточный программный код.

**// currPay автоматически приравнивается к 0.0F через значения,**

**// заданные по умолчанию.**

public Employee(string fullName) {

 this.fullName = fullName;

 this.empID = IDGenerator.GetNewEmpID();

}

Следует понимать, что использование ключевого слова this для передачи вызовов конструктора не является обязательным. Однако при использовании этого подхода вы получаете более удобное и более краткое определение класса. Фактически, используя этот подход, вы можете упростить свои программистские задачи, поскольку реальная работа делегируется одному конструктору (обычно это конструктор, который имеет наибольшее число параметров), а остальные конструкторы просто перекладывают ответственность.

**Определение открытого интерфейса класса**

После создания данных внутреннего состояния класса и набора конструкторов следующим шагом должно быть определение деталей *открытого интерфейса*класса. Этим терминам обозначают множество членов, непосредственно доступных из объектной переменной через операцию, обозначаемую точкой.

С точки зрения построения класса открытый интерфейс формируют те элементы, которые объявлены в классе с использованием ключевого слова public. Кроме далей данных и конструкторов, открытый интерфейс класса может иметь множество других членов, включая следующие.

• *Методы.*Именованные единицы действия, моделирующие некоторые особенности поведения класса.

• *Свойства.*Традиционные функции чтения и модификации данных.

• *Константы/поля только для чтения.*Поля данных, которые не допускают изменения после присваивания им значений (см. главу 3).

**Замечание**. Из последующего материала данной главы вам станет ясно, что вложенные определения типа тоже могут появляться в рамках открытого интерфейса типа. К тому же*,*как будет показано в главе 8, открытый интерфейс класса может поддерживать и события.

С учетом того, что наш класс Employee определяет два открытых метода (GiveBonus() и DisplayStats()), мы имеем возможность взаимодействовать с открытым интерфейсом так, как показано ниже.

**// Взаимодействие с открытым интерфейсом класса Employee.**

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Тип Employee в процессе работы \*\*\*\n");

 Employee e = new Employee("Джо", 80, 30000);

 e.GiveBonus(20.0);

 e.DisplayStats();

 Employee e2;

 e2 = new Employee("Бет", 81, 50000);

 e2.GiveBonus(1000);

 e2.DisplayStats();

 Console.ReadLine();

}

Если выполнить это приложение в том его состоянии, в котором оно находится сейчас, вы должны получить вывод, подобный показанному на рис. 4.2.

На этот момент у нас имеется очень простой тип класса с минимальным открытым интерфейсом. Перед тем как двигаться дальше к более сложным примерам, давайте потратим некоторое время на обсуждение принципов объектно-ориентированного программирования (рассмотрение типа Emplоуее мы продолжим немного позже).

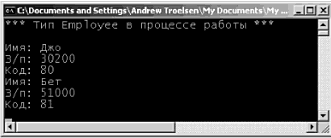


Рис. 4.2. Тип класса Employee в процессе работы

**Принципы объектно-ориентированного программирования**

Все объектно-ориентированные языки используют три базовых принципа объектно-ориентированного программирования.

• *Инкапсуляция*. Как данный язык скрывает внутренние особенности реализации объекта?

• *Наследование.*Как данный язык обеспечивает возможность многократного использования программного кода?

• *Полиморфизм.*Как данный язык позволяет интерпретировать родственные объекты унифицированным образом?

Перед тем как начать рассмотрение синтаксических особенностей реализации каждого из этих принципов, важно понять базовую роль каждого из них. Поэтому здесь предлагается краткая теоретическая информация по соответствующим вопросам, просто для того, чтобы исключить разночтения, которые могут возникать у людей, ощущающих постоянную нехватку времени на теорию из-за жестких сроков окончания их проектов.

**Инкапсуляция**

Первым принципом ООП является *инкапсуляция.*По сути, она означает возможность скрыть средствами языка несущественные детали реализации от пользователя объекта. Предположим, например, что мы используем класс DatabaseReader, который имеет два метода Open() и Close().

**// DatabaseReader инкапсулирует средства работы с базой данных.**

DatabaseReader dbObj = new DatabaseReader();

dbObj.Open(@"C:\Employees.mdf");

// **Работа с базой данных.**..

dbObj.Close();

Вымышленный класс DatabaseReader инкапсулирует внутренние возможности размещения, загрузки, обработки и закрытия файла данных. Пользователи объекта приветствуют инкапсуляцию, поскольку этот принцип ООП позволяет упростить задачи программирования. Нет необходимости беспокоиться о многочисленных строках программного кода, который выполняет работу класса DatabaseReader "за кулисами". Bсe, что требуется от вас, – это создание экземпляра и отправка подходящих сообщений (например, "открыть файл Employees.mdf, размещенный на моем диске C").

Одним из аспектов инкапсуляции является защита данных. В идеале данные состояния объекта должны определяться, как *приватные,*a не *открытые*(как было в предыдущих главах). В этом случае "внешний мир" будет вынужден "смиренно просить" право на изменение или чтение соответствующих значений.

**Наследование**

Следующим принципом ООП является наследование, означающее способность языка обеспечить построение определений новых классов на основе определений существующих классов. В сущности, наследование позволяет расширить возможности поведения базового класса (называемого также *родительским классом*) с помощью построения подкласса (называемого *производным классам*или *дочерним классом*), наследующего функциональные возможности родительского класса. На рис. 4.3 иллюстрируется отношение подчиненности ("is-а") для родительских и дочерних классов.

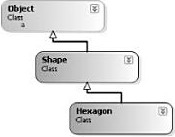


Рис. 4.3. Отношение подчиненности для родительских и дочерних классов

Можно прочитать эту диаграмму так: "Шестиугольник (hexagon) является формой (shape), которая является объектом (object)". При создании классов, связанных этой формой наследования, вы создаете отношения подчиненности между типами. Отношение подчиненности часто называется *классическим наследованием.*

Вспомните из главы 3, что System.Object является предельным базовым классом любой иерархии .NET. Здесь класс Shape (форма) расширяет Object (объект). Можно предположить, что Shape определяет некоторый набор свойств, полей, методов и событий, которые будут общими для всех форм. Класс Hexagon (шестиугольник) расширяет Shape и наследует функциональные возможности, определенные в рамках Shape и Object, вдобавок к определению своих собственных членов (какими бы они ни были).

В мире ООП есть и другая форма многократного использования программного кода - это модель локализации/делегирования (также известная, как отношение локализации, "has-a"). Эта форма многократного использования программного кода не используется дли создания отношений "класс-подкласс". Скорее данный класс может определить член-переменную другого класса и открыть часть или все свои функциональные возможности для "внешнего мира".

Например, если создается модель автомобиля, то вы можете отобразить тот факт, что автомобиль "имеет" ("has-a") радио. Было бы нелогично пытаться получить класс Car (автомобиль) из Radio (радио) или наоборот. (Радио является автомобилем? Я думаю, нет.) Скорее, есть два независимых класса, работающие вместе, где класс-контейнер создает и представляет функциональные возможности содержащегося в нем класса.

public class Radio {

 public void Power(bool turnOn) { Console.WriteLine("Radio on: {0}", turnOn); }

}

public class Car {

 // **Car содержит ("has-a") Radio.**

 private Radio myRadio = new Radio();

 public void TurnOnRadio(bool onOff) {

**// Делегат для внутреннего объекта.**

  myRadio.Power(onOff);

 }

}

Здесь тип-контейнер (Car) несет ответственность за создание содержащегося объекта (Radio). Если объект Car "желает" сделать поведение Radio доступным для экземпляра Car, он должен пополнить свой собственный открытый интерфейс некоторым набором функций, Которые будут действовать на содержащийся тип. Заметим, что пользователь объекта не получит никакой информации о том, что класс Car использует внутренний объект Radio.

static void Main(string[] args) {

 // **Вызов внутренне передается Radio.**

 Car viper = new Car();

 viper.TurnOnRadio(true);

}

**Полиморфизм**

Третьим принципом ООП является *полиморфизм.*Он характеризует способность языка одинаково интерпретировать родственные объекты. Эта особенность объектно-ориентированного языка позволяет базовому классу определить множество членов (формально называемых *полиморфным интерфейсом*) для всех производных классов. Полиморфный интерфейс типа класса строится с помощью определения произвольного числа *виртуальных,*или *абстрактных*членов. Виртуальный член *можно*изменить (или, говоря более формально, переопределить) в производном классе, тогда как абстрактный метод *должен*переопределяться. Когда производные типы переопределяют члены, определенные базовым классом, они, по существу, переопределяют свой ответ на соответствующий запрос.

Чтобы проиллюстрировать понятие полиморфизма, снова используем иерархию форм. Предположим, что класс Shape определил метод Draw(), не имеющий параметров и не возвращающий ничего. С учетом того, что визуализация для каждой формы оказывается уникальной, подклассы (такие как Hexagon и Circle) могут переопределить соответствующий метод так, как это требуется для них (рис. 4.4).

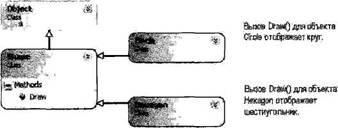


Рис. 4.4. Классический полиморфизм

После создания полиморфного интерфейса можно использовать различные предположения, касающиеся программного кода. Например, если Hexagon и Circle являются производными от одного общего родителя (Shape)*,*то некоторый массив типов Shape может содержать любой производный класс. Более того, если Shape определяет полиморфный интерфейс для всех производных типов {в данном примере это метод Draw(), то можно предположить, что каждый член в таком массиве имеет эти функциональные возможности. Проанализируйте следующий метод Main(), в котором массиву типов, производных от Shape, дается указание визуализировать себя с помощью метода Draw().

static void Main(string [] args) {

**// Создание массива элементов, производных от Shape.**

 Shape[] myShapes = new Shape[3];

 myShapes[0] = new Hexagon();

 myShapes[1] = new Circle();

 myShapes[2] = new Hexagon();

 // **Движение по массиву и отображение элементов.**

 foreach (Shape s in myShapes) s.Draw();

 Console.ReadLine();

}

На этом наш краткий (и упрощенный) обзор принципов ООП завершается. Теперь, имея в запасе теорию, мы исследуем некоторые подробности и точный синтаксис C#, с помощью которого реализуются каждый из указанных принципов.

**Первый принцип: сервис инкапсуляции C#**

Понятие инкапсуляции отражает общее правило, согласно которому поля данных объекта не должны быть непосредственно доступны из открытого интерфейса. Если пользователь объекта желает изменить состояние объекта, то он должен делать это косвенно, с помощью методов чтения (get) и модификации (set). В C# инкапсуляция "навязывается" на уровне синтаксиса с помощью ключевых слов public, private, protected и protected internal, как было показано в главе 3. Чтобы проиллюстрировать необходимость инкапсуляции, предположим, что у нас есть следующее определение класса.

**// Класс с одним общедоступным полем.**

public class Book {

**public**int numberOfPages;

}

Проблема общедоступных полей данных заключается в том, что такие элементы не имеют никакой возможности "понять", является ли их текущее значение действительным с точки зрения "бизнес-правил" данной системы. Вы знаете, что верхняя граница диапазона допустимых значений для int в C# очень велика (она равна 2147483647). Поэтому компилятор не запрещает следующий вариант присваивания.

**// М-м-м-да…**

static void Main(stting[] args) {

 Book miniNovel = new Book();

 miniNovel**.numberOfPages**= 30000000;

}

Здесь нет перехода за границы допустимости для данных целочисленного типа, но должно быть ясно, что miniNovel ("мини-роман") со значением 30000000 для numberOfPages (число страниц) является просто невероятным с практической точки зрения. Как видите, открытые поля не обеспечивают проверку адекватности данных. Если система предполагает правило, по которому мини-роман должен содержать от 1 до 200 страниц, будет трудно реализовать это правило программными средствами. В этой связи открытые поля обычно не находят места на уровне определений классов, применяемых для решения реальных задач (исключением являются открытые поля, доступные только для чтения).

Инкапсуляция обеспечивает возможность сохранения целостности данных для состояния объекта. Вместо определения открытых полей (с помощью которых очень просто прийти к нарушению целостности данных), вашей привычкой должно стать определение *частных полей данных,*которые обрабатываются вызывающей стороной косвенно, с использованием одного из двух главных подходов.

• Определение пары традиционных методов чтения и модификации данных.

• Определение именованного свойства.

Суть любого их этих подходов заключается в том, что хорошо инкапсулированный класс должен скрыть от "любопытных глаз внешнего мира" необработанные данные и детали того, как выполняются соответствующие действия. Часто такой подход называют *программированием "черного ящика".*Красота этого подхода в том, что создатель класса имеет возможность изменить особенности реализации любого из своих методов, так сказать, "за кулисами", и нет никакой нужды отменять использование уже существующего программного кода (при условии, что имя метода остается прежним).

**Инкапсуляция на основе методов чтения и модификации**

Давайте снова вернемся к рассмотрению нашего класса Employee. Чтобы "внешний мир" мог взаимодействовать с частным полем данных fullName, традиции велят определить средства *чтения* (метод get) и *модификации*(метод set). Например:

**// Традиционные средства чтения и модификации для приватных данных.**

public class Employee {

**private**string fullName;

 …

**// Чтение.**

 public string **GetFullName**() {return fullName;}

 // **Модификация**.

 public void **SetFullName**(string n) {

**// Удаление недопустимых символов (!, @, #, $, %)*,***

**// проверка максимальной длины (или регистра символов)**

**// перед присваиванием.**

  fullName = n;

 }

}

Конечно, компилятору "все равно", что вы будете вызывать методы чтения и модификации данных. Поскольку GetFullName() и SetFullName() инкапсулируют приватную строку с именем fullName, выбор таких имен кажется вполне подходящим. Логина вызова может быть следующей.

// **Использование средств чтения/модификации.**

static void Main(string[] args) {

 Employee p = new Employee();

 p.**SetFullName**("Фред Флинстон");

 Console.WriteLine("Имя работника: {0} ", p.**GetFullName**());

 Console.ReadLine();

}

**Инкапсуляция на основе свойств класса**

В отличие от традиционных методов чтения и модификации, языки .NET тяготеют к реализации принципа инкапсуляции на основе использования *свойств,*которые представляют доступные для внешнего пользователя элементы данных. Вместо того, чтобы вызывать два разных метода (get и set) для чтения и установки данных состояния объекта, пользователь получает возможность вызвать нечто, похожее на общедоступное поле. Предположим, что мы имеем свойство с именем ID (код), представляющее внутренний член-переменную empID типа Employee. Синтаксис вызова в данном случае должен выглядеть примерно так.

**// Синтаксис установки/чтения значения ID работника.**

static void Main(string[] args) {

 Employee p = new Employee();

 // **Установка значения.**

 p.**ID**= 81;

 // **Чтение значения.**

 Console.WriteLine ("ID работника: {0} ", p.**ID**); Console.ReadLine();

}

Свойства типа "за кадром" всегда отображаются в "настоящие" методы чтения и модификации. Поэтому, как разработчик класса, вы имеете возможность реализовать любую внутреннюю логику, выполняемую перед присваиванием соответствующего значения (например, перевод символов в верхний регистр, очистку значения от недопустимых символов, проверку принадлежности числового значения диапазону допустимости и т.д.). Ниже демонстрируется синтаксис C#. использующий, кроме свойства ID, свойство Pay (оплата), которое инкапсулирует поле currPay, a также свойство Name (имя), которое инкапсулирует данные fullName.

**// Инкапсуляция с помощью свойств**.

public class Employee {

 ...

 private int empID;

 private float currPay;

 private string fullName;

**// Свойство для empID.**

 public int ID {

**get**{return empID;}

**set**{

**// Вы можете проверить и, если требуется, модифицировать**

**// поступившее значение перед присваиванием.**

   empID = value;

  }

 }

**// Свойство для fullName.**

 public string Name {

  get { return fullName; }

  set { fullName = value; }

 }

**// Свойство для currPay.**

 public float Pay {

  get { return currPay; }

  set { currPay = value; }

 }

}

Свойство в C# компонуется из блока чтения и блока модификации (установки) значении. Ярлык value в C# представляет правую сторону оператора присваивания. Соответствующий ярлыку value тип данных зависит от того, какого сорта данные этот ярлык представляет. В данном примере свойство ID оперирует с типом данных int, который, как вы знаете, отображается в System.Int32.

**// 81 принадлежит System.Int32,**

**// поэтому "значением" является System.Int32.**

Employee e = new Employee();

**e.ID**= 81;

Чтобы это доказать, предположим, что вы добавили следующий программный код для установки свойства ID.

**// Свойство для empID.**

public int ID {

 get {return empID;}

 set {

  Console.WriteLine("value является экземпляром {0} ", value.**GetType**());

  Console.WriteLine("Значение value: {0} ", value); empID = value;

 }

}

Выполнив приложение, вы должны увидеть вариант вывода, показанный на рис. 4.5.



Рис. 4.5. Значение value после установки для ID значения 81

**Замечание**. Строго говоря, ярлык value в C# является не ключевым оловом, а, скорее, *контекстным ключевым словом,*представляющим неявный параметр, который используется в операторе присваивания в контексте метода, используемого для установки значения свойства. Поэтому вполне допустимо иметь члены-переменные и локальные элементы данных с именем value.

Следует понимать, что свойства (в отличие от традиционных методов чтения и модификации) еще и упрощают работу с типами, поскольку свойства способны "реагировать" на внутренние операции в C#. Например, предположим, что тип класса Employee имеет внутренний приватный член, представляющий значение возраста работника. Вот соответствующая модификация класса.

public class Employee {

 …

 // **Текущий возраст работника.**

 private int empAge;

 public Employee(string fullName, **int age,**int empID, float currPay) {

  …

**this.empAge**= **age;**

 }

 public int Age {

  get { return empAge; }

  set { empAge = value; }

 }

 public void DisplayStats() {

  …

**Console.WriteLine("Возраст: {0}**", **empAge);**

 }

}

Теперь предположим, что вы создали объект Employee с именем joe. Вы хотите, чтобы в день рождения работника значение переменной возраста увеличивалось на единицу. Используя традиционные методы чтения и модификации, вы должны применить, например, следующий программный код.

Employee joe = new Employee();

joe.**SetAge(**joe.**GetAge**() + 1);

Но если инкапсулировать empAge, используя "правильный" синтаксис, вы сможете просто написать:

Employee joe = new Employee();

joe.**Age**++;

**Внутреннее представление свойств в C#**

Многие программисты (особенно те, которые привыкли использовать C++) стремятся использовать традиционные префиксы get\_ и set\_ для методов чтения и модификации (например, get\_FullName() и set\_FullName()). Против самого соглашения возражений нет. Однако следует знать, что "за кадром" свойства в C# представляются программным кодом CIL, использующим такие же префиксы. Например, если открыть компоновочный блок Employees.exe с помощью ildasm.exe, вы увидите, что каждое свойство XXX на самом деле сводится к скрытым методам get\_XXX()/set\_XXX() (рис. 4.6).

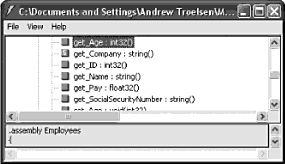


Рис. 4.6. Отображение свойств XXX в скрытые методы get\_XXX() и set\_XXX()

Предположим теперь, что тип Employee имеет частный член-переменную с именем empSSN для представления номера социальной страховки работника. Эта переменная устанавливается через параметр конструктора, а для управления этой переменной используется свойство SocialSecurityNumber.

**// Добавление поддержки нового поля, представляющего SSN-код.**

public class Employee {

 …

 // **Номер социальной страховки (SSN)**.

 private string empSSN;

 public Employes (string fullName, int age, int empID, float currPay, **string ssn)**{

  …

**this**.**empSSN**= **ssn**;

 }

 public string SocialSecurityNumber {

  get { return empSSN; }

  set { empSSN = value; }

 }

 public void DisplayStats() {

  …

**Console.WriteLine("SSN: {0} ", empSSN);**

 }

}

Если бы вы также определили два метода get\_SocialSecurityNumber() и set\_SocialSecurityNumber(), то получили бы ошибки компиляции.

**// Свойство в C# отображается в пару методов get\_/set\_.**

public class Employee {

**// ОШИБКА! Уже определены самим свойством!**

 public string get\_SocialSecurityNumber() { return empSSN; }

 public void set\_SocialSecurityNumber(string val) { empSSN = val; }

}

**Замечание.**В библиотеках базовых классов .NET всегда отдается предпочтение свойствам типа (в сравнении с традиционными методами чтения и модификации). Поэтому, чтобы строить пользовательские типы, которые хорошо интегрируются с платформой .NET, следует избегать использования традиционных методов get и set.

**Контекст операторов get и set для свойств**

До появления C# 2005 область видимости get и set задавалась исключительно модификаторами доступа в определении свойства.

**// Логика get и set здесь открыта,**

**// в соответствии с определением свойства.**

public string SocialSecurityNumber {

 get {return empSSN;}

 set {empSSN = value;}

}

В некоторых случаях бывает нужно указать свои области видимости для методов get и set. Чтобы сделать это, просто добавьте префикс доступности (в виде соответствующего ключевого слова) к ключевому слову get или set (при этом область видимости без уточнения будет соответствовать области видимости из определения свойства).

**// Пользователи объекта могут только получить значение,**

**// но производные типы могут также установить значение.**

public string SocialSecurityNumber {

 get { return empSSN;}

**protected**set {empSSN = value;}

}

В данном случае логика set для SocialSecurityNumber может вызываться только данным классом и *производными*классами, а поэтому не может быть доступна на уровне экземпляра объекта.

**Свойства, доступные только для чтения, и свойства, доступные только для записи**

При создании типов класса можно создавать свойства, доступные только для чтения. Для этого просто создайте свойство без соответствующего блока set. Точно так же, если вы хотите иметь свойство, допускающее только запись, опустите блок get. Для нашего примера в этом нет необходимости, но вот как можно изменить свойство SocialSecurityNumber, чтобы оно было доступно только для чтения.

public class Employee {

 …

**// Теперь это свойство, доступное только для чтения.**

 public string SocialSecurityNumber {**get**{return empSSN;}}

}

При таком изменений единственным способом установки номера социальной страховки для работника оказывается установка этого номера через аргумент конструктора.

**Статические свойства**

В C# также поддерживаются *статические свойства.*Вспомните из главы 3, что статические члены доступны на уровне класса, а не экземпляра (объекта) этого класса. Например, предположим, что тип Employee определяет элемент статических данных, представляющий название организации, в которой трудоустроен работник. Можно определить статическое свойство (например, уровня класса) так, как показано ниже.

**// Статические свойства должны оперировать со статическими данными!**

public class Employee {

 private **static**string companyName;

 public **static**String Company {

  get { return companyName; }

  set { companyName = value; }

 }

 …

}

Статические свойства используются точно так же, как статические методы.

**// Установка и чтение названия компании,**

**// в которой трудоустроены эти работники…**

public static int Main(string[] args) {

 Employee.Company = "Intertech training";

 Console.WriteLine("Эти люди, работают в {0} ", Employee.**Company**);

 …

}

Также вспомните из главы 3, что в C# поддерживаются статические конструкторы. Поэтому, если вы хотите, чтобы статическое свойство companyName всегда устанавливалось равным Intertech Training, можете добавить в класс Employee член следующего вида.

**// Статический конструктор без модификаторов доступа и аргументов.**

public class Employee {

 …

 static Employee() {

  companyName = "Intertech Training";

 }

}

В данном случае мы ничего не выиграли в результате добавления статического конструктора, если учесть, что тот же результат можно было бы достичь с помощью простого присваивания значения члену-переменной companyName, как показано ниже.

**// Статические свойства должны оперировать со статическими данными!**

public class Employee {

 private **static**string companyName = "Intertech Training";

}

Однако следует вспомнить о том. что статические конструкторы оказываются очень полезными тогда, когда значения для статических данных становятся известны только в среде выполнения (например, при чтении из базы данных).

В завершение нашего обзора возможностей инкапсуляции следует подчеркнуть, что свойства используются с той же целью, что и классическая пара методов чтения/модификации данных. Преимущество свойств заключается в том, что пользователи объекта получают возможность изменять внутренние данные, используя для этого один именованный элемент.

**Второй принцип: поддержка наследования в C#**

Теперь, после исследования различных подходов, позволяющих создавать классы с хорошей инкапсуляцией, пришло время заняться построением семейств связанных классов. Как уже упоминалось, наследование является принципом ООП, упрощающим многократное использование программного кода. Наследование бывает двух видов: классическое наследование (отношение подчиненности, "is-a") и модель локализации/делегирования (отношение локализации, "has-a"). Сначала мы рассмотрим классическую модель отношения подчиненности.

При создании отношения подчиненности между классами вы строите зависимость между типами. Основной идеей классического наследования является то, что новые классы могут использовать (и, возможно, расширять) функциональные возможности исходных классов. Для примера предположим, что вы хотите использовать функциональные возможности класса Employee и создать два новых класса – Salesperson (продавец) и Manager (менеджер). Иерархия классов будет выглядеть примерно так, как показано на рис. 4.7.

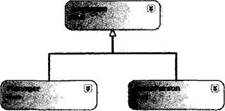


Рис. 4.7. Иерархия классов служащих

Из рис. 4.7 можно понять, что Salesperson (продавец) является ("is-a") Employee (работником), точно так же, как и Manager (менеджер). В классической модели наследования базовые классы (например. Employee) используются для определения общих характеристик, которые будут присущи всем потомкам. Подклассы (например, SalesPerson и Manager) расширяют общие функциональные возможности, добавляя специфические элементы поведения.

Для нашего примера мы предположим, что класс Manager расширяет Employee, обеспечивал запись числа опционов, а класс SalesPerson поддерживает информацию о числе продаж. В C# расширение класса задается в определении класса операцией, обозначаемой двоеточием (:). Так получаются производные типы класса в следующем фрагменте программного кода.

**// Добавление двух подклассов в пространстве имен Employees.**

namespace Employees {

 public class Manager: **Employee**{

**// Менеджер должен знать число опционов**.

  private ulong numberOfOptions;

  public ulong NumbOpts {

   get {return numberOfOptions;}

   set {numberOfOptions = value;}

  }

 }

 public class SalesPerson: **Employee**{

**// Продавец должен знать число продаж.**

private int numberOfSales;

  public int NumbSales {

   get {return numberOfSales;}

   set {numberOfSales = value;}

  }

 }

}

Теперь, когда создано отношение подчиненности, SalesPerson и Manager автоматически наследуют все открытие (и защищенные) члены базового класса Employee. Например**:**

// **Создание подкласса и доступ к функциональным возможностям**

// **базового класса.**

static void Main (string[] args) {

 // **Создание экземпляра SalesPerson.**

 SalesPerson stan = new SalesPerson();

 // **Эти члены наследуют возможности базового класса Employee.**

 stan.ID = 100;

 stan.Name = "Stan";

 // **Это определено классом SalesPerson**.

 stan.NumbSales = 42;

 Console.ReadLine();

}

Следует знать, что наследование сохраняет инкапсуляцию. Поэтому производный класс не может иметь непосредственный доступ к приватным членам, определенным базовым классом.

**Управление созданием базовых классов с помощью base**

В настоящий момент SalesPerson и Manager можно создать только с помощью конструктора, заданного по умолчанию. Поэтому предположим, что в тип Manager добавлен новый конструктор с шестью аргументами, который вызывается так, как показано ниже.

static void Main(string[] args) {

**// Предположим, что есть следующий конструктор с параметрами**

**// (имя, возраст, ID, плата, SSN, число опционов)**.

 Manager chucky = new Manager("Chucky", 35, 92, 100000, "333-23-2322", 9000);

}

Если взглянуть на список аргументов, можно сразу понять, что большинство из них должно запоминаться в членах-переменных, определенных базовым классом Employee. Для этого вы могли бы реализовать этот конструктор так, как предлагается ниже.

**// Если не указано иное, конструктор подкласса автоматически вызывает**

**// конструктор базового класса, заданный по умолчанию.**

public Manager(string fullName, int age, int empID, float currPay, string ssn, ulong numbOfOpts) {

**// Это наш элемент данных.**

 numberOfOptions = numbOfOpts;

 // **Использование членов, наследуемых от Employee,**

**// для установки данных состояния.**

 ID = empID;

 Age = age;

 Name = fullName;

 SocialSecurityNumber = ssn;

 Pay = currPay;

}

Строго говоря, это допустимый, но не оптимальный вариант. В C#, если вы не укажете иное, конструктор базового класса, *заданный по умолчанию,*вызывается автоматически до выполнения логики любого пользовательского конструктора Manager. После этого текущая реализация получает доступ к множеству открытых свойств базового класса Employee, чтобы задать его состояние. Поэтому здесь при создании производного объекта вы на самом деле "убиваете семь зайцев" (пять наследуемых свойств и два вызова конструктора)!

Чтобы оптимизировать создание производного класса, вы должны реализовать свои конструкторы подклассов так, чтобы явно вызвался подходящий пользовательский конструктор базового класса, а не конструктор, заданный по умолчанию. Таким образом можно уменьшить число вызовов инициализации наследуемых членов (что экономит время). Позвольте для этого модифицировать пользовательский конструктор.

**// На этот раз используем ключевое слово C# "base" для вызова**

**// пользовательского конструктора с базовым классом.**

public Manager (string fullName, int age, int empID, float currPay, string ssn, ulong numbOfOpts): **base(fullName, age, empID, currPay, ssn)**{

 numberOfOptions = numbOfOpts;

}

Здесь конструктор был дополнен довольно запутанными элементами синтаксиса. Непосредственно после закрывающей скобки списка аргументов конcтруктора стоит двоеточие, за которым следует ключевое слово C# base. В этой ситуации вы явно вызываете конструктор с пятью аргументами, определенный классом Employees избавляясь от ненужных вызовов в процессе создания дочернего класса.

Конструктор SalesPerson выглядит почти идентично.

**// Как правило, каждый подкласс должен явно вызывать**

**// подходящий конструктор базового класса**.

public SalesPerson(string fullName, int age*,*int empID, float currPay, string ssn, int numbOfSales): **base(fullName, age, empID, currPay, ssn)**{

 numberOfSales = numbOfSales;

}

Вы должны также знать о том. что ключевое слово base можно использовать всегда, когда подклассу нужно обеспечить доступ к открытым или защищенным членам, определённым родительским классом. Использование этого ключевого слова не ограничено логикой конструктора. В нашем следующем обсуждении полиморфизма вы увидите примеры, в которых тоже используется ключевое слово base.

**Множественные базовые классы**

Говоря о базовых классах, важно не забывать, что в C# каждый класс должен иметь *в точности один* непосредственный базовый класс. Таким образом, нельзя иметь тип с двумя или большим числом базовых классов (это называется множественным наследованием). Вы увидите в главе 7, что в C# любому типу позволяется иметь любое число дискретных интерфейсов. Таким образом класс в C# может реализовывать различные варианты поведения, избегая проблем, присущих классическому подходу, связанному с множественным наследованием. Аналогично можно сконфигурировать обычный интерфейс, как производный от множественных интерфейсов (см. главу 7).

**Хранение семейных тайн: ключевое слово protected**

Вы уже знаете, что открытые элементы непосредственно доступны отовсюду, а приватные элементы недоступны для объектов вне класса, определяющего эти элементы. Язык C#, занимающий лидирующие позиции среда многих других современных объектных языков, обеспечивает еще один уровень доступа: защищенный.

Если базовый класс определяет защищенные данные или защищенные члены, он может создать набор элементов, которые будут непосредственно доступны для любого дочернего класса. Например, чтобы позволить дочерним классам SalesPerson и Manager непосредственный доступ к сектору данных, определенному классом Employee, обновите оригинальное определение класса Employee так, как показано ниже.

**// Защищенные данные состояния.**

public class Employee {

**// Дочерние классы могут иметь непосредственный доступ**

**// к этой информации, а пользователи объекта – нет.**

**protected**string fullName;

**protected**int empID;

**protected**float currPay;

**protected**string empSSN;

**protected**int empAge;

 …

}

Удобство определения защищенных членов в базовом классе заключается в том, что производные типы теперь могут иметь доступ к данным не только через открытые методы и свойства. Недостатком, конечно, является то, что при прямом доступе производного типа к внутренним данным родителя вполне вероятно случайное нарушение существующих правил, установленных в рамках общедоступных свойств (например, превышение допустимого числа страниц для "мини-романа"). Создавая защищенные члены, вы задаете определенный уровень доверия между родительским и дочерним классом, поскольку компилятор не сможет обнаружить нарушения правил, предусмотренных вами для данного типа.

Наконец, следует понимать, что с точки зрения пользователя объекта защищенные данные воспринимаются, как приватные (поскольку пользователь находится "за пределами семейства"). Так, следующий вариант программного кода некорректен.

static void Main(string[] args) {

**// Ошибка! Защищенные данные недоступны на уровне экземпляра.**

 Employee emp = new Employee();

**emp.empSSN**= "111-11-1111";

}

**Запрет наследования: изолированные классы**

Создавая отношения базовых классов и подклассов, вы получаете возможность использовать поведение существующих типов. Но что делать, когда нужно определить класс, не позволяющий получение подклассов? Например, предположим, что в наше пространство имен добавлен еще один класс, расширяющий уже существующий тип SalesPerson. На рис. 4.8 показана соответствующая схема.

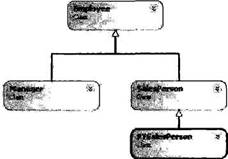


Рис. 4.8. Расширенная иерархия служащих

Класс PTSalesPerson является классом, представляющим продавца, работающего на неполную ставку, и предположим, например, что вы хотите, чтобы никакой другой разработчик не мог создавать подклассы из PTSalesPerson. (В конце концов, какую еще неполную ставку можно получить на основе неполной ставки?) Чтобы не допустить возможности расширения класса, используйте ключевое слово C# sealed.

**// Класс PTSalesPerson не сможет быть базовым классом.**

public **sealed** class PTSalesPerson: SalesPerson {

 public PTSalesPerson(string fullName, int age, int empID, float currPay, string ssn, int numbOfSales): base (fullName*,*age, empID, currPay, ssn, numbOfSales) {

  // **Логика конструктора…**

 }

**// Другие члены…**

}

Поскольку класс PTSalesPerson изолирован, он не может служить базовым Классам никакому другому типу. Поэтому при попытке расширить PTSalesPersоn вы получите сообщение об ошибке компиляции.

**// Ошибка компиляции!**

public class ReallyPTSalesPerson: **PTSalesPerson**{…}

Наиболее полезным ключевое слово sealed оказывается при создании автономных классов утилит. Класс String, определённый в пространстве имен Sуstem, например, явно изолирован.

public **sealed**class string: object, IComparable, ICloneable, IConvertible, IEnumerable {…}

Поэтому вы не сможете создать новый класс, производный от System.String:

// **Снова ошибка!**

public class MyString: string

Если вы хотите создать новый класс, использующий функциональные возможности изолированного класса, единственным вариантом будет отказ от классического наследования и использование модели локализации/делегирования (известной еще как отношение локализации, "has-a").

**Модель локализации/делегирования**

Как уже отмечалось в этой главе, наследование можно реализовать двумя способами. Только что мы исследовали классическое отношение подчиненности ("is-a"). Чтобы завершить обсуждение второго принципа ООП, давайте рассмотрим отношение локализации (отношение "has-a", также известное под названием модели *локализации/делегирования).*Предположим, что мы создали новый класс, моделирующий пакет льгот работника.

**// Этот тип будет функционировать, как вложенный класс.**

public class BenefitPackage {

**// Другие члены, представляющие пакет страховок,**

**// медицинского обслуживания и т.д.**

 public double ComputePayDeduction() {return 125.0;}

}

Ясно, что отношение подчиненности ("is-a") между типами BenefitPackage (пакет льгот) и Employee (работник) выглядело бы достаточно странно. (Является ли менеджер пакетом льгот? Вряд ли.) Но должно быть ясно и то, что какая-то связь между этими типами необходима. Короче, вы должны выразить ту идею, что каждый работник имеет ("has-a") пакет льгот. Для этого определение класса Employee следует обновить так", как показано ниже.

**// Работники теперь имеют льготы.**

public class Employee {

 …

 // **Содержит объект BenefitPackage**.

 protected BenefitPackage empBenefits = new BenefitPackage();

}

Здесь вы успешно создали вложенный объект. Но чтобы открыть функциональные возможности вложенного объекта внешнему миру, требуется *делегирование.*Делегирование означает добавление в класс-контейнер таких членов, которые будут использовать функциональные возможности содержащегося в классе объекта. Например, можно изменить класс Employee так, чтобы он открывал содержащийся в нем объект empBenefits с помощью некоторого свойства, а также позволял использовать функциональные возможности этого объекта с помощью нового метода GetBenefitCost().

public class Employee {

 protected BenefitPackage empBenefits = new BenefitPackage();

 // **Открытие некоторых функциональных возможностей объекта.**

 public double GetBenefitCost() {

  return empBenefits.ComputePayDeduction();

 }

 // **Доступ к объекту через пользовательское свойство.**

 public BenefitPackage Benefits {

  get {return empBenefits;}

  set {empBenefits = value;}

 }

}

В следующем обновленном методе Main() обратите внимание на то, как можно взаимодействовать с внутренним типом BenefitsPackage, определяемым типом Employee.

static void Main(string[] args) {

 Manager mel;

 mel = new Manager();

 Console.WriteLine (mel.**Benefits**.ComputerPayDeduction());

…

 Consolе.ReadLine();

}

**Вложенные определения типов**

Перед тем как рассмотреть заключительный принцип ООП (полиморфизм), давайте обсудим технику программирования, называемую *вложением типов*. В C# можно определить тип (перечень, класс, интерфейс, структуру или делегат) в пределах области видимости класса или структуры. При этом вложенный ("внутренний") тип считается *членом*содержащего его ("внешнего") класса, с точки зрения среды выполнения ничем не отличающимся от любого другого члена (поля, свойства, метода, события и т.п.). Синтаксис, используемый для вложения типа, исключительно прост.

public class OuterClass {

**// Открытый вложенный тип могут использовать все.**

 public class PublicInnerClass{}

**// Приватный вложенный тип могут использовать только члены**

**// содержащего его класса.**

 private class PrivateInnerClass{}

}

Синтаксис здесь ясен, но понять, *почему*вы можете это делать, не так просто. Чтобы прийти к пониманию этого подхода, подумайте над следующим.

• Вложение типов подобно их композиции ("has-a"), за исключением того, что вы имеете полный контроль над доступом на уровне внутреннего *типа,*а не содержащегося объекта.

• Ввиду того, что вложенный тип является членом класса-контейнера, этот тип может иметь доступ к приватным членам данного класса.

• Часто вложенный тип играет роль вспомогательного элемента для класса-контейнера, и его использование "внешним миром" не предполагается.

Когда некоторый тип содержит другой тип класса, он может создавать члены-переменные вложенного типа, точно так же, как для любого другого своего элемента данных. Но если вы пытаетесь использовать вложенный тип из-за границ внешнего типа, вы должны уточнить область видимости вложенного типа, указав его абсолютное имя. Изучите следующий пример программного кода.

static void Main (string[] args) {

**// Создание и использование открытого внутреннего класса. Все ОК!**

 OuterClass.PublicInnerClass inner;

 inner = new OuterClass.PublicInnerClass();

**// Ошибка компиляции! Нет доступа к приватному классу.**

 OuterClass.PrivateInnerClass inner2;

 inner2 = new OuterClass.PrivateInnerClass();

}

Чтобы использовать этот подход в нашем примере, предположим, что мы вложили BenefitPackage непосредственно в тип класса Employee.

**// Вложение BenefitPackage.**

public class Employee {

 ...

 public class BenefitPackage {

  public double ComputePayDeduction() {return 125.0;}

 }

}

Глубина вложения может быть любой. Предположим, что мы хотим создать перечень BenefitPackageLevel, указывающий различные уровни льгот, которые может выбрать работник. Чтобы программно реализовать связь между Employee, BenefitPackage и BenefitPackageLevel, можно вложить перечень так, как показано ниже.

**// Employee содержит BenefitPackage**.

public class Employee {

 // **BenefitPackage содержит BenefitPackageLevel.**

 public class BenefitPackage {

  public double ComputePayDeduction() {return 125.0;}

  public enum BenefitPackageLevel {

   Standard, Gold, Platinum

  }

 }

}

С учетом отношений вложении обратите внимание на то, как приходится иcпользовать этот перечень.

Static void Main(string[] args) {

**// Создание переменной BenefitPackageLevel**.

 Employee.BenefitPackage.BenefitPackageLevel myBenefitLevel = Employee.BenefitPackage.BenefitPackageLevel.Platinum;

 …

}

**Третий принцип: поддержка полиморфизма в C#**

Теперь давайте рассмотрим заключительный принцип ООП – полиморфизм. Напомним, что базовый класс Employee определил метод GiveBonus(), который был реализован так.

**// Предоставление премий работникам.**

public class Employee {

 …

 public void GiveBonus(float amount) { currPay += amount;}

}

Ввиду того, что этот метод является открытым, вы теперь имеете возможность выдать премии продавцам и менеджерам (а также продавцам, занятым неполный рабочий день).

static void Main(string[] args) {

 // **Премии работникам.**

 Manager chucky = new Manager("Сhucky", 50, 92, 100000, "333-23-2322", 9000);

 chucky.**GiveBonus**(300);

 chucky.DisplayStats();

 SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32*-*3232 " *,*31);

 fran.**GiveBonus**(200); fran.DisplayStats();

 Console.ReadLine();

}

Недостаток данного варианта в том, что наследуемый метод GiveBonus() действует одинаково для всех подклассов. В идеале премия продавца, как и продавца, работающего на неполную ставку, должна зависеть от числа продаж. Возможно, менеджеры должны получать льготы в дополнение к денежному вознаграждению, В связи с этим возникает интересный вопрос: "Каким образом родственные объекты могут по-разному реагировать на одинаковые запросы?"

**Ключевые слова virtual и override**

Полиморфизм обеспечивает подклассам возможность задать собственную реализацию методов, определенных базовым классом. Чтобы соответствующим образом изменить наш проект, мы должны рассмотреть применение ключевых слов C# virtual и override. Если в базовом классе нужно определить метод, *допускающий*переопределение подклассом, то этот метод должен быть виртуальным.

public class Employee {

**// GiveBonus() имеет реализацию, заданную по умолчанию,**

**// но дочерние классы могут переопределить это поведение.**

 public **virtual** void GiveBonus(float amount) {currPay += amount;}

 …

}

Чтобы в подклассе переопределить виртуальный метод, используется ключевое слово override. Например, SalesPerson и Manager могут переопределить GiveBonus() так, как показано ниже (мы предполагаем, что PTSalesPerson переопределяет GiveBonus() примерно так же, как SalesPerson),

public class SalesPerson: Employee {

**// Премия продавца зависит от числа продаж.**

 public **override**void GiveBonus(float amount) {

  int salesBonus = 0;

  if (numberOfSales ›= 0 && numberOfSales ‹= 100) salesBonus = 10;

  else

   if (numberOfSales ›= 101&& numberOfSales ‹= 200) salesBonus = 15;

   else salesBonus = 20; // **Вcе, что больше 200.**

**base**.GiveBonus(amount \* salesBonus)*;*

}

}

public class Manager: Employee {

 // **Менеджер в дополнение к денежному вознаграждению**

**// получает некоторое число опционов.**

public **override**void GiveBonus(float amount) {

**// Прибавка к зарплате**.

  base.GiveBonus(amount);

**// И получение опционов…**

  Random r = new Random();

  numberOfOptions += (ulong)r.Next(500);

 }

 …

}

Обратите внимание на то, что переопределенный метод будет использовать поведение, принятое по умолчанию, если указать ключевое слово base. При этом нет необходимости снова реализовывать логику GiveBonus(), а можно снова использовать (и, возможно, расширить) поведение родительского класса, принятое по умолчанию.

Также предположим, что Employee.DisplayStats() был объявлен виртуально и переопределен каждым подклассом, чтобы учесть число продаж (для продавцов) и текущее состояние опционов (для менеджеров). Теперь, когда каждый подкласс может по-своему интерпретировать виртуальные методы, каждый экземпляр объекта ведет себя более независимо.

static void Main (string[] args) {

**// Лучшая система премиальных!**

 Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);

 chucky.**GiveBonus**(300);

 chucky.DisplayStats();

 SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);

 fran.**GiveBonus**(200);

 fran.DisplayStats();

}

**Снова о ключевом слове sealed**

Ключевое слово sealed может также применяться к членам типа, чтобы запретить переопределение таких виртуальных членив в производных типах. Это оказывается полезным тогда, когда нужно изолировать не весь класс, а только несколько его методов или свойств.

Например, если (по некоторой причине) классу PTSalesPerson требуется разрешить расширение другими классами, но нужно гарантировать, чтобы эти классы *не могли*переопределять виртуальный метод GiveBonus(), можно использовать следующий вариант программного кода.

**// Этот класс можно расширить,**

**// но GiveBonus() не может переопределяться производным классом.**

public class PTSalesPerson: SalesPerson {

 …

 public override sealed void GiveBonus(float amount) {

  …

 }

}

**Абстрактные классы**

В данный момент базовый класс Employee скомпонован так, что он может поставлять своим потомкам защищенные члены-переменные, а также два виртуальных метода (GiveBonus() и DisplayStats()), которые могут переопределяться производным классом. Все это хорошо, но данный вариант программного кода имеет один недостаток: вы можете непосредственно создавать экземпляры базового класса Employee.

**// Что же это значит?**

Employee X = new Employee ();

В данном примере единственной целью базового класса Employee является определение общих полей и членов для всех подклассов. Вероятно, вы не предполагали, что кто-то будет непосредственно создавать экземпляры класса, поскольку тип Employee (работник) является слишком общим. Например, если я приду к вам и скажу "Я работаю!", то в ответ я, скорее всего, услышу вопрос *"Кем*вы работаете?" (консультантом, инструктором, ассистентом администратора, редактором, представителем Белого Дома и т.п.).

Ввиду того, что многие базовые классы оказываются чем-то вроде "небожителей", для нашего примера лучше всего *запретить*возможность непосредственного создания новых объектов Employee. В C# это можно сделать программными средствами, используя ключевое слово abstract.

**// Обозначение класса Employee, как абстрактного,**

**// запрещает непосредственное создание его экземпляров.**

**abstract** public class Employee {…}

Если вы теперь попытаетесь создать экземпляр класса Employee, то получите ошибку компиляции.

**// Ошибка! Нельзя создать экземпляр абстрактного класса.**

Employee X = new Employee();

Превосходно! К этому моменту мы построили очень интересную иерархию служащих. Мы добавим новые функциональные возможности в это приложение немного позже, когда будем рассматривать правила классификации в C#. Иерархия типов, определенных на данный момент, показана на рис. 4.9.

**Исходный код.**Проект Employees размещен в подкаталоге, соответствующем главе 4.

**Принудительный полиморфизм: абстрактные методы**

Если класс является абстрактным базовым классом, он может определять любое число *абстрактных членов*(их аналогами в C++ являются "чистые" виртуальные функции). Абстрактные методы могут использоваться тогда, когда требуется определить метод *без*реализации, заданной по умолчанию. В результате производным классам придется использовать полиморфизм, поскольку им придется "уточнять" детали абстрактных методов. Здесь сразу же возникает вопрос, зачем это нужно. Чтобы понять роль абстрактных методов, давайте снова рассмотрим иерархию форм, уже упоминавшуюся в этой главе и расширенную так, как показано на рис. 4.10.

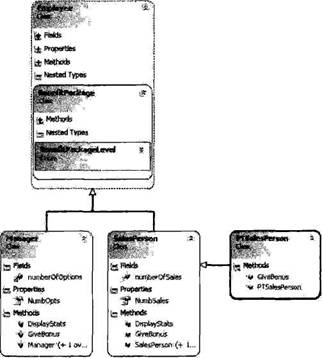


Рис. 4.9. Полная иерархии служащих



Рис. 4.10. Иерархия форм

Как и в случае иерархии служащих, лучше запретить пользователю объекта создавать экземпляры Shape (форма) непосредственно, поскольку соответствующее понятие слишком абстрактно. Для этого необходимо определить тип Shape, как абстрактный класс.

namespace Shapes {

 public **abstract**class Shape {

  // **Форме можно назначить понятное имя.**

protected string petName;

  // **Конструкторы.**

  public Shape()(petName = "БезИмени";}

  public Shape(string s) (petName = s;}

  // **Draw() виртуален и может быть переопределен.**

  public **virtual**void Draw() {

   Console.WriteLine("Shape.Draw()");

  }

  public string PetName {

   get { return petName; }

   set { petName = value; }

  }

 }

 // **Circle не переопределяет Draw()**.

 public class Circle: Shape {

  public Circle() {}

  public Circle(string name): base(name) {}

 }

 // **Hexagon переопределяет Draw()**.

 public class Hexagon: Shape {

  public Hexagon () {}

  public Hexagon (string name): base(name) {}

  public **override**void Draw() {

   Console.WriteLine("Отображение шестиугольника {0}", petName);

  }

 }

}

Обратите внимание на то, что класс Shape определил виртуальный метод с именем Draw(). Вы только что убедились, что подклассы могут переопределять поведение виртуального метода, используя ключевое слово override (как в случае класса Hexagon). Роль абстрактных методов становится совершенно ясной, если вспомнить, что подклассам не обязательно переопределять виртуальные методы (как в случае Circle). Таким образом, если вы создадите экземпляры типов Hexagon и Circle, то обнаружите, что Hexagon "знает", как правильно отобразить себя. Однако Circle в этом случае будет "не на шутку озадачен" (рис. 4.11).

**// Объект Circle не переопределяет реализацию Draw() базового класса**.

static void Main(string[] args) {

 Hexagon hex = new Hexagon("Beth");

 hex.Draw();

 Circle car = new Circle("Cindy");

 // **М-м-м-да. Используем реализацию базового класса.**

 cir.Draw();

 Console.ReadLine();

}



Рис. 4.11. Виртуальные методы переопределять не обязательно

Ясно, что это не идеальный вариант иерархии форм. Чтобы заставить каждый производный класс иметь свой собственный метод Draw(), можно задать Draw(), как абстрактный метод класса Shape, т.е метод, который вообще не имеет реализации, заданной по умолчанию. Заметим, что абстрактные методы могут определяться только в абстрактных классах. Если вы попытаетесь сделать это в другом классе, то получите ошибку компиляции.

**// Заставим всех "деток" иметь cвоe представление.**

public **abstract** class Shape {

 ...

**// Теперь Draw() полностью абстрактный**

**// (обратите внимание на точку с запятой)**.

 public **abstract**void Draw();

 …

}

Учитывая это, вы обязаны реализовать Draw() в классе Circle. Иначе Circle тоже должен быть абстрактным типом, обозначенным ключевым словом abstract (что для данного примера не совсем логично).

**// Если не задать реализацию метода Draw(), то класс Circle должен**

**// быть абстрактным и не допускать непосредcтвенную реализацию!**

public class Circle: Shape {

 public Circle() {}

 public Circle(string name): base (name) {}

 // **Теперь Circle должен "понимать"**, **как отобразить себя.**

 public **override**void Draw() {

  Console.WriteLine("Отображение окружности {0}", petName);

 }

}

Для иллюстрации упомянутых здесь возможностей полиморфизма рассмотрим следующий программный код,

// **Создание массива различных объектов Shape.**

static void Main(string [] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с полиморфизмом \*\*\*\*\*\n");

 Shape[] myShapes = {new Hexagon(), new Circle(), new Hexagon("Mick"), new Circle("Beth"), new Hexagon("Linda")};

 // **Движение по массиву и отображение объектов.**

 for (int i = 0; i ‹ myShapes.Length; i++) myShapes[i].Draw();

 Console.ReadLine();

}

Соответствующий вывод показан на рис. 4.12.

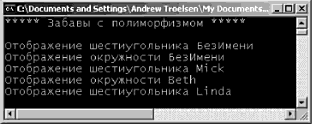


Рис. 4.12. Забавы с полиморфизмом

Здесь метод Main() иллюстрирует полиморфизм в лучшем его проявлении. Напомним, что при обозначении класса, как абстрактного, вы теряете возможность *непосредственного создания экземпляров*соответствующего типа. Однако вы можете хранить ссылки на любой подкласс в абстрактной базовой переменной. При выполнении итераций по массиву ссылок Shape соответствующий тип будет определен в среде выполнения. После этого будет вызван соответствующий метод.

**Возможность скрывать члены**

В C# также обеспечивается логическая противоположность возможности переопределения методов; возможность скрывать члены. Формально говоря, если производный класс повторно объявляет член, идентичный унаследованному от базового класса, полученный класс *скрывает* (или затеняет) соответствующий член родительского класса. На практике эта возможность оказывается наиболее полезной тогда, когда приходится создавать подклассы, созданные другими разработчиками (например, при использовании купленного пакета программ .NET).

Для иллюстрации предположим, что от своего коллеги (или одноклассника) вы получили класс ThreeDCircle, который получается из System.Object.

public class ThreeDCircie {

 public void Draw() {

  Console.WriteLine ("Отображение трехмерной окружности");

 }

}

Вы полагаете, что ThreeDCircle относится ("is-a") к типу Circle, поэтому пытаетесь получить производный класс из существующего типа Circle.

public class ThreeDCircie: Circle {

 public void Draw() {

  Console.WriteLine("Отображение трехмерной окружности");

 }

}

В процессе компиляции в Visual Studio 2005 вы увидите предупреждение, показанное на рис. 4.13. ('Shapes.ThreeDCircle.Draw()' скрывает наследуемый член 'Shapes.Circle.Draw()'. Чтобы переопределить соответствующую реализацию данным членом, используйте ключевое слово override, иначе используйте ключевое слово new)

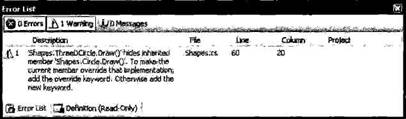


Рис. 4.13. Ой! ThreeDCircle.Draw() скрывает Circle.Draw

Есть два варианта решения этой проблемы. Можно просто изменить версию Draw() родителя, используя ключевое слово override. При таком подходе тип ThreeDCircie может расширить возможности поведения родителя так, как требуется.

Альтернативой может быть использование ключевого слова new (с членом Draw() типа ThreeDCircle). Это явное указание того, что реализация производного типа должна скрывать версию родителя (это может понадобиться тогда, когда полученные извне программы .NET не согласуются с программами, уже имеющимися у вас).

**// Этот класс расширяет Circle и скрывает наследуемый метод Draw()**.

public class ThreeDCircle: Circle {

**// Скрыть любую внешнюю реализацию Draw()**.

 public **new**void Draw() {

  Console.WriteLine("Отображение трехмерной окружности");

 }

}

Вы можете использовать ключевое слово new с любыми членами, унаследованными от базового класса (с полями, константами, статическими членами, свойствами и т.д.). Например, предположим, что ThreeDCircle должен скрыть наследуемое поле petName.

public class ThreeDCircle: Circle {

**new**protected string petName;

**new**public void Draw() {

  Console.WriteLine("Отображение трехмерной окружности");

 }

}

Наконец, следует знать о том, что возможность использовать реализацию базового класса для скрытого члена тоже не исключается, но в этом случае следует использовать явное приведение типов (см. следующий раздел). Например:

static void Main(string[] args) {

 ThreeDCircie о = new ThreeDCircle();

 о.Draw(); // **Вызывается ThreeDCircle.Draw**()

 ((Circle)o).Draw(); // **Вызывается Circle.Draw()**

}

**Исходный код.**Иерархия Shapes размещается в подкаталоге, соответствующем главе 4.

**Правила приведения типов в C#**

Пришло время изучить правила выполнения *операций приведения типов*в C#. Вспомните иерархию Employees и тот факт, что наивысшим классом в системе является System.Object. Поэтому все в вашей программе является объектами и может рассматриваться, как объекты. С учетом этого вполне допустимо сохранять экземпляры любого типа в объектных переменных.

// **Manager – это System.Object.**

object frank = new Manager("Frank Zappa", 9, 40000, "111-11-1111", 5);

В системе Employees типы Manager, Salesperson и PTSalesPerson расширяют Employee, поэтому можно запомнить любой из этих объектов в подходящей ссылке базового класса. Так что допустимыми будут и следующие операторы.

**// Manager - это Employee.**

Employee moonUnit = new Manager("MoonUnit Zappa", 2, 20000, "101-11-1321", 1);

// **PTSalesPerson - это Salesperson.**

Salesperson jill = new PTSalesPerson("Jill", 834, 100000, "111-12-1119", 90);

Первым правилом преобразований для типов классов является то, что когда два класса связаны отношением подчиненности ("is-a"), всегда можно сохранить производный тип в ссылке базового класса. Формально его называют *неявным (кон-текстуальным) приведением типов,*поскольку оно "работает" только в рамках законов наследования.

Такой подход позволяет строить очень мощные программные конструкции. Предположим, например, что у нас есть класс TheMachine (машина), который поддерживает следующий статический метод, соответствующий увольнению работника.

public class TheMachine {

 public static void FireThisPerson(Employee e) {

**// Удалить из базы данных…**

**// Забрать у работника ключи и точилку…**

 }

}

Мы можем непосредственно передать этому методу любой производный класс класса Employee ввиду того, что эти классы связаны отношением подчиненности ("is-a").

**// Сокращение штатов.**

TheMaсhine.FireThisPerson(moonUnit); *//***"moonUnit" - это Employee.**

TheMachine.FireThisFerson(jill); //**"jill" - это SalesPerson.**

В дальнейшем программный код использует в производном типе неявное преобразование из базового класса (Employee). Но что делать, если вы хотите уволить служащего по имени Frank Zарра (информация о котором в настоящий момент хранится в ссылке System.Object общего вида)? Если передать объект frank непосредственно в TheMaсhine.FireThisPerson() так, как показано ниже:

**// Manager - это object, но…**.

object frank = new Manager("Frank Zappa", 9, 40000, "111-11-1111", 5);

…

TheMachine.FireThisPerson(frank); // **Ошибка!**

то вы получите ошибку компиляции. Причина в том, что нельзя автоматически интерпретировать System.Object, как объект, являющийся производным непосредственно от Employee, поскольку Object не является Employee. Но вы можете заметить, что объектная ссылка указывает на объект, связанный с Employee. Поэтому компилятор будет "удовлетворен", если вы используете *явное приведение типов.*

В C# явное приведение типов указывается с помощью скобок, размещаемых вокруг имени типа, к которому вы хотите прийти, с последующим указанием объекта, который вы пытаетесь использовать в качестве исходного. Например:

**// Приведение общего типа System.Object**

**// к строго типизованному Manager.**

Manager mgr = (Manager)frank;

Console.WriteLine("Опционы Фрэнка: {0}", mgr.NumbOpts);

Если не объявлять специальную переменную для "целевого типа", то можно получить более компактный программный код.

**// "Внутристрочное" явное приведение типов.**

Console.WriteLine("Опционы Фрэнка: {0}", ((Manager)frank).NumbOpts);

Проблему, связанную с передачей ссылки System.Object методу FireThisPerson(), можно решить так, как показано ниже.

**// Явное приведение типа System.Object к Employee.**

TheMachine.FireThisPerson((Employee)frank);

**Замечание.**Попытка приведения объекта к несовместимому типу порождает в среде выполнения соответствующее исключение. Возможности структурированной обработки исключений будут рассмотрены в главе 6.

**Распознавание типов**

Статический метод TheMachine.FireThisPerson() строился так, чтобы он мог принимать любой тип, производный от Employee, но возникает один вопрос: как метод "узнает", какой именно производный тип передается методу. Кроме того, если поступивший параметр имеет тип Employee, то как получить доступ к специфическим членам типов SalesPerson и Manager?

Язык C# обеспечивает три способа определения того, что ссылка базового класса действительно указывает на производный тип: явное приведение типа (рассмотренное выше), ключевое слово is и ключевое слово as. Ключевое слово is возвращает логическое значение, указывающее на совместимость ссылки базового класса с данным производным типом. Рассмотрим следующий обновленный метод FireThisPerson().

public class TheMachine {

 public static void FireThisPerson(Employee e) {

  if **(e is SalesPerson)**{

   Console.WriteLine("Имя уволенного продавца: {0}", e.GetFullName());

   Console.WriteLine("{0} оформил(a) {1} операций…", e.GetFullName(), ((SalesPerson)e).NumbSales);

  }

  if **(e is Manager)**{

   Console.WriteLine("Имя уволенного клерка: {0}", e.GetFullName());

   Console.WriteLine("{0} имел(а) (1} опцион(ов)…", e.GetFullName(), ((Manager)e).NumbOpts);

  }

 }

}

Здесь ключевое слово is используется для того, чтобы динамически определить тип работника. Чтобы получить доступ к свойствам NumbSales или NumbOpts, вы должны использовать явное приведение типов. Альтернативой место бы быть ис-пользование ключевого слова as для получения ссылки на производный тип (если типы при этом окажутся несовместимыми, ссылка получит значение null).

SalesPerson p = е as SalesРеrson;

if (p!= null) Console.WriteLinе("Число продаж: {0}", p.NumbSales);

Замечание. Из Главы 7 вы узнаете, что такой же подход (явное приведение типов, is и as) может использоваться при получении интерфейсных ссылок из реализующего типа.

**Приведение числовых типов**

В завершение нашего обзора операций приведения типов в C# заметим, что преобразование числовых типов подчиняется примерно таким же правилам. Чтобы поместить "больший" числовой тип в "меньший" (например, целое число int в byte), следует использовать явное приведение типов, которое информирует компилятор о том, что вы готовы принять возможную потерю данных.

**// Если "х" больше предельного значения для byte, вероятна потеря**

**// данных, но из главы 9 вы узнаете о "контролируемых исключениях",**

**// с помощью которых можно управлять результатом.**

int х = 6;

byte b = (**byte**)x;

Когда вы сохраняете "меньший" числовой тип в "большем" (например, byte в int), тип для вас будет преобразован неявно и автоматически, так как здесь нет потерь данных.

**// Приведение типа не требуется,**

**// int достаточно "велик" для хранения byte.**

byte b = 30; int x = b;

**Парциальные типы C#**

В C# 2005 вводится новый модификатор типа partial, который позволяет определять C#-тип в нескольких файлах \*.cs. Предыдущие версии языка C# требовали, чтобы весь программный код определения типа содержался в пределах одного файла \*.cs. С учетом того, что C#-класс производственного уровня может содержать сотни строк программного кода, соответствующий файл может оказаться достаточно объемным.

В таких случаях было бы хорошо иметь возможность разделить реализацию типа на несколько файлов, чтобы отделить программный код, который в некотором смысле более важен, от других элементов. Например, используя для класса модификатор partial, можно поместить все открытые члены в файл с именем MyТуре\_Public.cs, а приватные поля данных и вспомогательные функции – в файл MyType\_Private.cs.

**// MyClass\_Public.cs**

namespace PartialTypes {

 public **partial**class MyClass {

  // **Конструкторы.**

public MyClass() {}

  // **Открытые члены.**

  public void MemberA() {}

  public void MemberB() {}

 }

}

**// MyClass\_Private.cs**

namespace PartialTypes {

 public **partial**class MyClass {

  // **Приватные поля данных.**

private string someStringData;

  // **Приватные вспомогательные члены.**

  public static void SomeStaticHelper(){}

 }

}

Это, в частности, упростит задачу изучения открытого интерфейса типа для новых членов команды разработчиков. Вместо изучения единого (и большого) файла C# с целью поиска соответствующих членов, разработчики получают возможность рассмотреть только открытые члены. Конечно, после компиляции этих файлов с помощью csc.exe в результате все равно получается единый унифицированный тип (рис. 4.14).

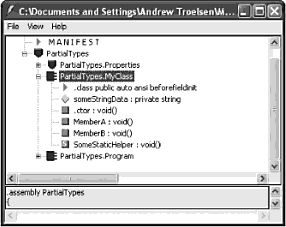


Рис. 4.14. После компиляции парциальные типы уже не будут парциальными

**Исходный код.**Проект PartialTypes размещен в подкаталоге, соответствующем главе 4.

**Замечание**. После рассмотрения Windows Forms и ASP.NET вы поймете, что в Visual Studio 2005 ключевое слово partial используется для разделения программного кода, генерируемого инструментами разработки. Используя этот подход, вы можете сосредоточиться на поиске подходящих решений и не заботиться об автоматически генерируемом программном коде.

**Документирование исходного кода в C# с помощью XML**

В завершение этой главы мы рассмотрим специфические для C# лексемы комментариев, которые порождают документацию программного кода на. базе XML. Если вы имеете опыт программирования на языке Java, то, скорее всего, знаете об утилите javadoc. Используя javadoc, можно превратить исходный вод Java в соответствующее HTML-представление. Модель документирования, принятая в C#, оказывается немного иной в том отношении, что процесс преобразования комментариев в XML является заботой компилятора (при использовании опции /doc), а не особой утилиты.

Но почему для документирования определений типов используется XML, а не HTML? Главная причина в том, что XML обеспечивает очень широкие возможности. Поскольку XML отделяет определение данных от представления этих данных, к лежащему в основе XML-коду можно применить любое XML-преобразование, позволяющее отобразить документацию программного кода в одном из множества доступных форматов (MSDN, HTML и т.д.).

При документировании C#-типов в формате XML первой задачей является выбор одного из двух вариантой нотации: тройной косой черты (///) или признака комментария, который начинается комбинацией косой черты и двух звездочек (/\*\*), а заканчивается – комбинацией звездочки и косой черты (\*/). В поле документирующего комментария можно использовать любые XML-элементы, включая элементы рекомендуемого набора, описанные в табл. 4.1.

Таблица 4.1. Элементы XML рекомендуемые для использования в комментариях к программному коду

|  |  |
| --- | --- |
| **XML-элемент документации** | **Описание** |
| ‹с› | Указывает текст, который должен отображаться  "шрифтом для программного кода" |
| ‹code› | Указывает множество строк, которое должно рассматриваться,  как программный код |
| <example> | Указывает пример программного кода для описываемого элемента |
| ‹exception› | Документирует возможные исключения данного класса |
| ‹list› | Вставляет список или таблицу в файл документации |
| ‹раrаm› | Описывает данный параметр |
| ‹paramref› | Ассоциирует данный дескриптор XML с параметром |
| <permission> | Документирует ограничения защиты для данного члена |
| ‹remarks› | Создает описание для данного члена |
| ‹returns› | Документирует возвращаемое значение данного члена |
| ‹see› | Перекрестная ссылка для связанных элементов документа |
| ‹seealso› | Создает раздел '"см. также" в описании |
| ‹summary› | Документирует "поясняющее резюме" для данного члена |
| ‹value› | Документирует данное свойство |

В качестве конкретного примера рассмотрим следующее определение типа Car (автомобиль), в котором следует обратить особое внимание на использование элементов ‹summary› и ‹param›.

**/// ‹summary›**

**/// Это тип Car, иллюстрирующий**

**/// возможности XML-документирования.**

**/// ‹/summary›**

public class Car {

 /// **‹summary›**

**/// Есть ли люк в крыше вашего автомобиля?**

**/// ‹/summary›**

 private bool hasSunroof = false;

 /// **‹summary›**

**/// Этот конструктор позволяет установить наличие люка.**

**/// ‹/summary›**

**/// ‹param name="hasSunroof "› ‹/param›**

 public Car(bool hasSunroof) {

  this.hasSunroof = hasSunroof;

 }

 /// **‹summary›**

**/// Этот метод позволяет открыть люк.**

**/// ‹/summary›**

**/// ‹param name="state"› ‹/param›**

 public void OpenSunroof (bool state) {

  if (state == true && hasSunroof == true) Console.WriteLine("Открываем люк!");

  else Console.WriteLine("Извините, у вас нет люка.");

 }

}

Метод Main() программы также документируется с использованием XML-элементов.

**/// ‹summary›**

**/// Точка входа приложения.**

**/// ‹/summary›**

static void Main(string [] args) {

 Car с = new Car(true);

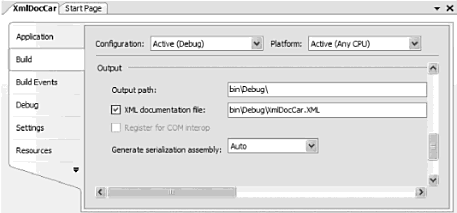
 с.OpenSunroof(true);

}

Чтобы на основе комментариев, задающих XML-код, сгенерировать соответствующий файл \*.xml, при построении C#-программы с помощью csc.exe используется флаг /doc.

csc /doc:XmlCarDoc.xml \*.cs

В Visual Studio 2005 можно указать имя файла с XML-документацией*,*используя вкладку Build окна свойств (рис. 4.15).



Pис. 4.15. Генерирование файла XML-документации в Visual Studio 2005

**Символы форматирования в XML-коде комментариев**

Если открыть сгенерированный XML-файл, вы увидите, что элементы будут помечены такими символами, как "M", "T", "F" и т.п. Например:

‹member name = "Т:ХmlDоcCar.Car"›

 ‹summary›

  Это тип Car, иллюстрирующий возможности XML-документирования.

 ‹/summary›

‹/member›

В табл. 4.2 описаны значения этих меток.

**Таблица 4.2.**Символы форматирования XML

|  |  |
| --- | --- |
| **Символ форматирования** | **Описание** |
| E | Элемент обозначает событие |
| F | Элемент представляет поле |
| M | Элемент представляет метод (включая конструкторы и перегруженные операции) |
| N | Элемент определяет пространство имен |
| P | Элемент представляет свойство типа (включая индексы) |
| T | Элемент представляет тип (например, класс, интерфейс, структуру, перечень, делегат) |

**Трансформация XML-кода комментариев**

Предыдущие версии Visual Studio 2005 (в частности. Visual Studio .NET 2003) предлагали очень полезный инструмент, позволяющий преобразовать файлы с XML-кодом документации в систему HTML-справки. К сожалению, Visual Studio 2005 не предлагает такой утилиты, оставляя пользователя "один на один" с XML-документом. Если вы имеете опыт использования XML-трансформаций, то, конечно, способны вручную создать подходящие таблицы стилей.

Более простым вариантом является использование инструментов сторонних производителей, которые позволяют переводить XML-код в самые разные форматы. Например, приложение NDoc, уже упоминавшееся в главе 2, позволяет генерировать документацию в нескольких различных форматах. Напомним, что информацию о приложении NDoc можно найти на страницах http://ndoc.sourceforge.net.

**Исходный код.**Проект XmlDocCar размещен в подкаталоге, соответствующем главе 4.

**Резюме**

Если вы изучаете .NET, имея опыт работы с любым другим объектно-ориентированным языком программирования, то материал этой главы обеспечит сравнение возможностей используемого вами языка с возможностями языка C#. При отсутствии такого опыта многие представленные в этой главе понятия могут казаться непривычными. Но это не страшно, поскольку по мере освоения оставшегося материала книги вы будете иметь возможность закрепить представленные здесь понятия.

Эта глава началась с обсуждения принципов ООП: инкапсуляции, наследования и полиморфизма. Сервис инкапсуляции можно обеспечить с помощью традиционных методов чтения/модификации, свойств типа или открытых полей, доступных только для чтения. Наследование в C# реализуется еще проще, поскольку этот язык не имеет для наследования специального ключевого слова, а предлагает использовать операцию, обозначаемую двоеточием. Наконец, для поддержки полиморфизма в C# предлагается использовать ключевые слова abstract, virtual, override и new.

**ГЛАВА 5. Цикл существования объектов**

В предыдущей главе мы потратили достаточно много времени на то, чтобы научиться строить пользовательские типы класса в C#. В этой главе мы выясним, как среда CLR управляет уже размешенными объектами с помощью процесса, который называется *сборкой мусора.*Программистам использующим C# не приходится удалять объекты из памяти "вручную" (напомним, что в C# вообще нет ключевого слова delete). Объекты .NET размещаются в области памяти, которая называется *управляемой динамической памятью,*где эти объекты "в некоторый подходящий момент" будут автоматически уничтожены сборщиком мусора.

Выяснив основные детали процесса сборки мусора, вы узнаете, как взаимодействовать со сборщиком мусора, используя для этого тип класса System.GC, Наконец, мы рассмотрим виртуальный метод System.Object.Finalize() и интерфейс IDisposable, которые можно использовать для того, чтобы создавать типы самостоятельно освобождающие в нужный момент свои внутренние неуправляемые ресурсы. Изучив материал этой главы, вы сможете понять, как среда CLR управляет объектами .NET.

**Классы, объекты и ссылки**

Чтобы очертить контуры темы, рассматриваемой в данной главе, необходимо уточнить различия между класcами, объектами и ссылками. В предыдущей главе уже говорилось о том, что класс – это своеобразный "шаблон" с описанием того, как экземпляр данного типа должен выглядеть и вести себя в памяти. Классы определяются в файлах, которые по соглашению в C# имеют расширение \*.cs. Рассмотрим простой класс Car (автомобиль), определённый в файле Car.cs.

public class Car {

 private int currSp;

 private string petName;

 public Car(){}

 public Car(string name, int speed) {

  petName = name;

  currSp = speed;

 }

 public override string ToString() {

  return String.Format("{0} имеет скорость {1} км/ч", petName, currSp);

 }

}

Определив класс, вы можете разместить в памяти любое число соответствующих объектов, используя ключевое слово C# new. При этом, однако, следует понимать, что ключевое слово new возвращает*ссылку*на объект в динамической памяти, а не сам реальный объект. Эта переменная со ссылкой запоминается в стеке для использования в приложении в дальнейшем. Для вызова членов объекта следует применить к сохраненной ссылке операцию C#, обозначаемую точкой.

class Program {

 static void Main(string[] args) {

  // **Создается новый объект Car в динамической памяти.**

**// Возвращается ссылка на этот объект ('refТоМуСаr**').

  Car refToMyCar = new Car("Zippy", 50);

  // **Операция C#, обозначаемая точкой (.), используется**

**// со ссылочной переменной для вызова членов этого объекта.**

  Console.WriteLine(refToMyCar.ToString());

  Console.ReadLine();

 }

}

На рис. 5.1 изображена схема, иллюстрирующая взаимосвязь между классами, объектами и их ссылками.

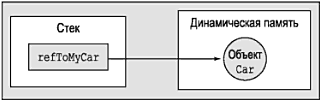


Рис. 5.1. Ссылки на объекты в управляемой динамической памяти

**Основные сведения о существовании объектов**

При построении C#-приложений вы вправе предполагать, что управляемая динамическая память будет обрабатываться без вашего прямого вмешательства. "Золотое правило" управления памятью .NET является очень простым.

• *Правило.*Следует поместить объект в управляемую динамическую память с помощью ключевого слова new и забыть об этом.

Сборщик мусора уничтожит созданный объект, когда этот объект будет больше не нужен. Cледующий очевидный вопрос: "Как сборщик мусора определяет, что объект больше не нужен?" Краткий (т.е. упрощенный) ответ заключается в том, что сборщик мусора удаляет объект из динамической памяти тогда, когда объект становится *недоступным*для всех частей программного кода. Предположим, что вы имеете метод, размещающий локальный объект Car.

public static void MakeACar() {

**// Если myCar является единственной ссылкой на объект Car,**

**// то объект может быть уничтожен после возвращения из метода.**

 Car myCar = new Car();

 …

}

Обратите внимание на то, что ссылка на объект (myCar) была создана непосредственно в методе MakeACar() и не передавалась за пределы области видимости определяющего эту ссылку объекта (ни в виде возвращаемого значения, ни в виде параметров ref/out). Поэтому после завершения работы вызванного метода ссылка myCar становится недоступной, и соответствующий объект Car оказывается*кандидатом*для удаления в "мусор". Однако следует понимать, что вы не можете гарантировать немедленное удаление этого объекта из памяти сразу же по завершении работы MakeACar(). В этот момент можно гарантировать только то, что при следующей сборке мусора в общеязыковой среде выполнения (CLR) объект myCar может быть без опасений уничтожен.

Вы, несомненно, обнаружите, что программирование в окружении, обеспечивающем автоматическую сборку мусора, значительно упрощает задачу разработки приложений. Программисты, использующие C++, знают о том, что если в C++ забыть вручную удалить размещенные в динамической памяти объекты, может произойти "утечка памяти". На самом деле ликвидация утечек памяти является одним из самых трудоемких (и неинтересных) аспектов программирования на языках, которые не являются управляемыми. Поручив сборщику мусора уничтожение объектов, вы снимаете с себя груз ответственности за управление памятью и перекладываете его на CLR.

Замечание. Если вы имеете опыт разработки программ в использованием COM, то знайте, что объекты .NET не поддерживают счетчик внутренних ссылок, поэтому управляемые объекты не предлагают такие методы, как AddRef() и Release().

**CIL-код для new**

Когда компилятор C# обнаруживает ключевое слово new, он генерирует CIL-инструкцию newobj в рамках реализации соответствующего метода. Если выполнить компиляцию программного кода текущего примера и с помощью ildasm.exe рассмотреть полученный компоновочный блок, то в рамках метода MakeACar() вы увидите следующие CIL-операторы.

.method public hidebysig static void MakeACar() cil managed

{

**// Code size 7 (0x7)**

 .maxstack 1

 .locals init ([0] class SimpleFinalize.Car c)

 IL\_0000: **newobj**instance void SimpleFinalize.Car::.ctor()

 IL\_0005: stloc.0

 IL\_0006: ret}

} // end of method Program::MakeACar

Перед тем как обсудить точные правила, определяющие момент удаления объекта из управляемой динамической памяти, давайте выясним роль CIL-инструкции newobj. Сначала заметим, что управляемая динамическая память является не просто случайным фрагментом памяти, доступной для среды выполнения. Сборщик мусора .NET является исключительно аккуратным "дворником" в динамической памяти – он (при необходимости) даже сжимает пустые блоки памяти с целью оптимизации. Чтобы упростить задачу сборки мусора, управляемая динамическая память имеет указатель (обычно называемый *указателем на следующий объект,*или *указателем на новый объект*),который идентифицирует точное место размещения следующего объекта.

Инструкция newobj информирует среду CLR о том, что необходимо выполнить следующие главные задачи.

• Вычислить общий объем памяти, необходимой для размещения объекта (включая память, необходимую для членов-переменных и базовых классов типа).

• Проверить управляемую динамическую память, чтобы гарантировать достаточный объем памяти для размещаемого объекта. Если памяти достаточно, вызывается конструктор типа, и вызывающей стороне, в конечном счете, возвращается ссылка на новый объект в памяти, причем адрес этого объекта будет соответствовать последней позиции указателя на следующий объект.

• Наконец, перед возвращением ссылки вызывающей стороне необходимо изменить значение указателя на следующий объект, чтобы указатель соответствовал началу свободной области управляемой динамической памяти.

Этот процесс схематически показан на риc. 5.2.

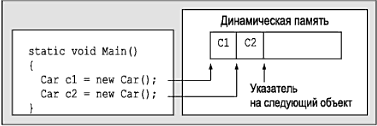


Рис. 5.2. Размещение объектов в управляемой динамической памяти

При размещении объектов в приложении пространство управляемой динамической памяти может, в конечном счете, заполниться. Если при обработке инструкции newobj среда CLR обнаружит, что управляемой динамической памяти недостаточно для размещения очередного типа, с целью освобождения памяти будет выполнена сборка мусора. Поэтому следующее правило сборки мусора также оказывается очень простым.

• *Правило.*Если управляемая динамическая память недостаточна дли размещения очередного объекта, выполняется сборка мусора.

В процессе сборки мусора сборщик мусора временно приостанавливает все активные потоки в рамках текущего процесса, чтобы гарантировать, что приложение не получит доступа к динамической памяти в процессе сборки мусора. Тему потоков мы рассмотрим в главе 14, а пока что воспринимайте поток, как "нить" выполнения в пределах выполняющейся программы. После завершения цикла сборки мусора приостановленным потокам будет разрешено продолжить работу. К счастью, сборщик мусора .NET хорошо оптимизирован, и вы вряд ли заметите соответствующие короткие перерывы в работе вашего приложения.

**Роль корней приложения**

Теперь мы вернемся к вопросу о том, как сборщик мусора определяет, когда объект "больше не нужен". Чтобы понять это, необходимо рассмотреть понятие *корней приложения.*Упрощенно говоря,*корень*- это место в памяти со ссылкой на объект, размещенный в динамической памяти. Строго говоря, корень может относиться к любой из следующих категорий.

• Ссылки на глобальные объекты (хотя они и не позволены в C#, программный код CIL допускает размещение глобальных объектов).

• Ссылки на используемый в настоящий момент статические объекты и поля.

• Ссылки на локальные объекты в пределах данного метода.

• Ссылки на объектные параметры, предаваемые методу.

• Ссылки на объекты, ожидающие *финализации*(соответствующее понятие будет описано в этой главе позже).

• Любые регистры процессора, ссылающиеся на локальный объект.

В процессе сборки мусора среда выполнения проверяет объекты в управляемой динамической памяти и определяет, остаются ли они доступными для приложения (иначе говоря, *укорененными*)*.*Для этого среда CLR строит *объектный граф,*изображающий каждый объект в динамической памяти, достижимый для приложения. Объектные графы будут рассматриваться подробнее при обсуждении сериализации объектов (см. главу 17). На данный момент вам достаточно знать, что объектные графы используются для учета всех достижимых объектов. Следует знать и о том, что сборщик мусора никогда не учитывает один и тот же объект дважды, вследствие чего, в отличие от классического подхода COM. здесь не возникает опасности циклического учета ссылок,.

Предположим, что управляемая динамическая память содержит множество объектов, имена которых A, B, C, D, E, F и G. В процессе сборки мусора эти объекты (а также все внутренние объектные ссылки, которые эти объекты могут содержать) проверяются на наличие у них активных корней, После построения графа недостижимые объекты (мы будем предполагать, что это объекты C и F) обозначаются, как мусор.

На рис. 5.3 представлен возможный объектный граф для только что описанного сценария (направленные стрелки, связывающие объекты в таком графе, можно заменить словами "зависит от" или "требует", – например, "E зависит от G и косвенно от B", "A не зависит ни от чего" и т.д.).

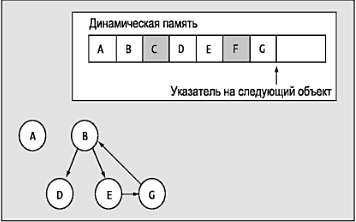


Рис. 5.3. Объектные графы строятся для выявления объектов, достижимых из корней приложения

Если объекты помечены для уничтожения (в данном случае это C и F – они не включены в объектный граф), эти объекты удаляются из памяти. В этот момент оставшееся пространство в динамической памяти уплотняется, что в свою очередь заставляет среду CLR модифицировать множество корней активного приложения, чтобы они обеспечивали правильные ссылки на точки размещения в памяти (это делается автоматически и незаметно). Наконец, соответствующим образом изменяется указатель на следующий объект. На рис. 5.4 показан результат преобразований.

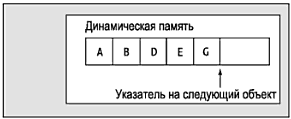


Рис. 5.4. "Чистая и компактная" динамическая память

**Замечание.**Строго говоря, сборщик мусора использует два разных фрагмента динамической памяти, один из которых предназначен специально для хранения очень больших объектов. К этой динамической памяти при сборке мусора система обращается реже, поскольку при размещении больших объектов возможные потери производительности выше. Тем не менее, вполне допустимо считать "управляемую динамическую память" единым регионом в памяти.

**Генерации объектов**

Когда среда CLR пытается найти недоступные объекты, это не значит, что будет рассмотрен буквально каждый объект, размещенный в управляемой динамической памяти. Очевидно, что это требовало бы слишком много времени, особенно в реальных (т.е. больших) приложениях.

Чтобы оптимизировать процесс, каждый объект в динамической памяти приписывается определенной "генерации". Идея достаточно проста: чем дольше объект существует в динамической памяти, тем более вероятно то, что он должен там и оставаться. Например, объект, реализующий Main() будет находиться в памяти до тех пор, пока программа не закончится. С другой стороны, объекты, которые недавно размещены в динамической памяти, вероятнее всего, станут вскоре недостижимыми (например, объекты, созданные в рамках области видимости метода). При этих предположениях каждый объект в динамической памяти можно отнести к одной из следующих категорий.

• *Генерация 0.*Новые, только что размещенные объекты, которые еще никогда же предназначались для использования в процессе сборки мусора.

• *Генерация 1.* Объекты, которые "пережили" одну сборку мусора (т.е. были обозначены для использования в процессе сборки мусора, но не были удалены по той причине, что в динамической памяти оказалось достаточно места).

• *Генерация*2. Объекты, ''пережившие" несколько сборок мусора.

Сборщик мусора сначала рассматривает объекты генерации 0*.*Если в результате выявления ненужных объектов и соответствующей чистки свободной памяти оказывается достаточно, все оставшиеся объекты относятся к генерации 1. Чтобы понять, как генерации объектов влияют на процесс сборки мусора, рассмотрите рис. 5.5. где схематически показано, как некоторое множество "выживших" объектов (A, B и E) генерации 0 переводятся в следующую генерацию после обновления остальной части памяти.



Рис 5.5. Объекты генерации 0, которые "пережили" сборку мусора, переходят к генерации 1

Если все объекты генерации 0 уже рассмотрены, но памяти все равно еще не достаточно, то рассматривается "достижимость" объектов генерации 1 и выполняется сборка мусора среди этих объектов. "Выжившие" объекты генерации 1 переходят к генерации 2. Если сборщик мусора *все еще*требует дополнительной памяти, тогда оцениваются объекты генерации 2. Здесь, если объект генерации 2 "выживает" в процессе сборки мусора, то такой объект сохраняет принадлежность к генерации 2, поскольку это предел для генераций объектов.

Итак, с помощью назначения признака генерации объектам в динамической памяти более новые объекты (например, локальные переменные) будут удаляться быстрее, тогда как старые объекты (такие как, например, объект приложения программы) будут "беспокоиться" значительно реже.

**Тип System.GC**

Библиотеки базовых классов предлагают тип класса System.GC, который позволяет программно взаимодействовать со сборщиком мусора, используя множество статических членов указанного класса. Следует заметить, что непосредственно использовать этот тип в программном коде приходится очень редко (если приходится вообще). Чаще всего члены типа System.GC используется тогда, когда создаются типы, использующие *неуправляемые ресурсы.*В табл. 5.1 предлагаются описания некоторых членов этого класса (подробности можно найти в документации .NET Framework 2.0 SDK).

**Таблица 5.1.**"Избранные" члены типа System.GC

|  |  |
| --- | --- |
| **Члены System.GC** | **Описание** |
| AddMemoryPressure(), RemoveMemoryPressure() | Позволяют указать числовое значение, характеризующее  "срочность" вызова процесса сборки мусора.  Эти методы должны изменять уровень "давления"  *согласованно*(в частности, удаляемая величина не  должна превышать добавленную) |
| Collect() | Вынуждает GC выполнить сборку мусора |
| CollectionCount() | Возвращает числовое значение, указывающее, сколько раз  "выживала" данная генерация при сборке мусора |
| GetGeneration() | Возвращает информацию о генерации, к которой в настоящий  момент относится объект |
| GetTotalMemory() | Возвращает оценку объема памяти (в байтах), выделенной  для управляемой динамической памяти в настоящий момент.  Логический параметр указывает, должен ли вызов ждать начала сборки мусора, чтобы возвратить результат |
| MaxGeneration | Возвращает максимум для числа генераций, поддерживаемых в системе. В Microsoft .NET 2.0, предполагается существование трех генераций (0, 1 и 2) |
| SuppressFinalize() | Устанавливает индикатор того, что данный объект не должен вызывать свой метод Finalize() |
| WaitForPendingFinalizers() | Приостанавливает выполнение текущего потока, пока не будут отработаны все объекты, предусматривающие финализацию. Этот метод обычно вызывается непосредственно после вызова GC.Collect() |

Рассмотрите следующий метод Main(), в котором иллюстрируется использование указанных членов System.GC.

static void Main(string[] args) {

**// Вывод оценки (в байтах) для динамической памяти.**

 Console.WriteLine("Оценка объема памяти (в байтах): {0}", GC.GetTotalMemory(false));

 // **Отсчет для MaxGeneration начинается с нуля,**

**// поэтому для удобства добавляем 1**.

 Console.WriteLine("Число генераций для данной OC: {0}\n", (GC.МахGeneration + 1));

 Car refToMyCar = new Car("Zippy", 100);

 Console.WriteLine(refToMyCar.ToString());

 // **Вывод информации о генерации для объекта refToMyCar**.

 Console.WriteLine("Генерация refToMyCar: {0}", GC.SetGeneration(refToMyCar));

 Console.ReadLine();

}

**Активизация сборки мусора**

Итак, сборщик мусора в .NET призван управлять памятью за вас. Однако в очень редких случаях, перечисленных ниже, бывает выгодно программно активизировать начало сборки мусора, используя дня этого GC.Collect().

• Перед входом приложения в блок программного кода, для которого нежелательно, чтобы его выполнение прерывалось возможной сборкой мусора.

• После окончания размещения очень большого числа объектов, когда вы желаете освободить как можно больше памяти.

Если вы сочтете, что будет выгодно выполнить сборку мусора, вы можете явно начать процесс сборки мусора так, как показано ниже.

static void Main(string[] args) {

 …

**// Активизация сборки мусора и**

**// ожидание завершения финализации объектов.**

 GC.Collect();

 GC.WaitForPendingFinalizers();

 …

}

При непосредственной активизации сборки мусора вы должны вызвать GC.WaitForPendingFinalizers(). В рамках этого подхода вы можете быть уверены, что все лредуcматривающие финализацию объекты обязательно получат возможность выполнить все необходимые завершающие действия, прежде чем ваша программа продолжит свою работу. "За кулисами" GC.WaitForPendingFinalizers() приостановит выполнение вызывающего "потока" на время процесса сборки мусора. Это гарантирует, что ваш программный код не сможет вызвать метод объекта, уничтожаемого в данный момент.

Методу GC.Collect() можно передать числовое значение, указывающее старейшую генерацию, для которой должна быть выполнена сборка мусора. Например, если вы желаете сообщить CLR, что следует рассмотреть только объекты генерации 0, вы должны напечатать следующее.

static void Main(string[] args) {

 …

**// Рассмотреть только объекты генерации 0.**

 GC.Collect(0);

 GC.WaitForPendingFinalizers();

 …

}

Подобно любой сборке мусора, вызов GC.Collect() повысит статус выживших генераций. Предположим, например, что наш метод Main() обновлен так, как показано ниже.

static void Main(string[] args) {

 Console.WriteLine ("\*\*\*\*\* Забавы с System.GC \*\*\*\*\*\n");

 // **Вывод информации об объеме динамической памяти.**

 Console.WriteLine("Оценка объёма памяти (в байтах): {0}", GC.GetTotalMemory(false));

 // **Отсчет для MaxGeneration начинается с нуля.**

 Console.WriteLine("Число генераций для данной ОС: {0}\n", (GC.MaxGeneration + 1));

 Car refToMyCar = new Car("Zippy", 100);

 Console.WriteLine(refToMyCar.ToString());

 // **Вывод информации о генерации для объекта refToMyCar.**

 Console.WriteLine("\nГенерация refToMyCar: {0}", GC.GetGeneration(refToMyCar));

 // **Создание тысяч объектов с целью тестирования.**

 object[] tonsOfObjects = new object[50000];

 for (int i = 0; i ‹ 50000; i++) tonsOfObjects [i] = new object();

 // **Сборка мусора только для объектов генерации 0.**

 GC.Collect(0);

 GC.WaitForPendingFinalizers();

 // **Вывод информации о генерации для объекта refToMyCar.**

 Console.WriteLine("Генерация refToMyCar: {0}", GC.GetGeneration(refToMyCar));

**// Проверим, "жив" ли объект tonsOfObjects[9000]**.

 if (tonsOfObjects[9000] != null) {

  Console.WriteLine("Генерация tonsOfObjects[9000]: {0}", GC.GetGeneration(tonsOfObjects[9000]));

 } else Console.WriteLine("Объекта tonsOfObjects[9000] ужe нет");

 // **Вывод числа процедур сборки мусора для генераций.**

 Console.WriteLine("\nДля ген. 0 сборка выполнялась {0}: раз(a)", GC.CollectionCount(0));

 Console.WriteLine("Для ген. 1 сборка выполнялась {0} раз(а)", GC.CollectionCount(1));

 Console.WriteLine("Для ген. 2 сборка выполнялась {0} раз(a)", GC.CollectionCount(2));

 Console.ReadLine();

}

Здесь, мы намерений создали очень большой массив объектов с целью тестирования. Как следует из вывода, показанного на рис. 5.6, хотя метод Main() делает всего один явный запрос на сборку мусора, среда CLR выполняет целый ряд операций сборки мусора в фоновом режиме.

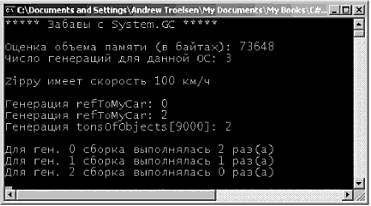


Рис. 5.6. Взаимодействие со сборщиком мусора CLR через System.GC

К этому моменту, я надеюсь, вы уяснили себе некоторые детали, касающиеся цикла существования объектов. Остаток этой главы мы посвятим дальнейшему изучению процесса сборки мусора, выяснив, как можно строить объекты, предусматривающие финализацию, и объекты, предусматривающие освобождение ресурсов. Следует заметить, что обсуждающийся ниже подход может быть полезен только при построении управляемых классов с поддержкой внутренних неуправляемых ресурсов.

**Исходный код**. Проект SimpleGC размещён в подкаталоге, соответствующем главе 5.

**Создание объектов, предусматривающих финализацию**

В главе 3 говорилось о том, что главный базовый класс .NET, System.Object, определяет виртуальный метод с именем Finalize() (метод деструктора). Реализация этого метода, заданная по умолчанию, не делает ничего.

// **System.Object**

public class Object {

 …

 protected virtual void Finalize(){}

}

Переопределяя Finalize() в своем пользовательском классе, вы создаете программную логику "уборки", необходимую для вашего типа. Поскольку этот член определяется, как protected, непосредственно вызвать метод Finalized объекта будет невозможно, Метод Finalize () объекта вызывается сборщиком мусора перед удалением объекта из памяти (если, конечно, этот метод объектом поддерживается).

Ясно, что обращение к Finalize() происходит и в процессе "естественной" сборки мусора, и в случае программной активизации сборки мусора с помощью GC.Collect(). Кроме того, метод деструктора типа будет автоматически вызван тогда, когда выгружается из памяти *домен приложения,*содержащий выполняемое приложение. Вы, возможно, знаете, что домены приложений используются для размещения выполняемого компоновочного блока и необходимых для него внешних библиотек программного кода. Если вы еще не знакомы с этим понятием .NET, то всю необходимую информацию вам предоставит глава 13. Здесь главное то, что при выгрузке из памяти домена приложения среда CLR автоматически вызывает деструкторы для каждого из предусматривающих финализацию объектов, созданных в процессе выполнения программы.

Теперь, независимо от того, что может говорить вам интуиция разработчика, следует подчеркнуть, что большинству классов в C# не требуется никакой явной "уборки". Причина проста: если ваши типы используют другие управляемые объекты, то все, в конечном счете, будет обработано сборщиком мусора. Создавать класс, который должен заниматься "уборкой", вам придется только тогда, когда этот класс будет использовать *неуправляемые ресурсы*(например, прямой доступ к дескрипторам файлов ОС, неуправляемым базам данных или другим неуправляемым ресурсам). Вы, наверное, знаете, что неуправляемые ресурсы создаются в результате прямого вызова API операционной системы с помощью PInvoke (Platform Invocation – обращение к платформе) или с помощью некоторых довольно сложных сценариев взаимодействия COM. С учетом этого возникает следующее правило сборки мусора.

• *Правило.*Необходимость переопределения Finalize() может возникать только тогда, когда класс C# использует неуправляемые ресурсы посредством PInvoke или при решении сложных задач взаимодействия с COM-объектами (обычно с применением типа System.Runtime.InteropServices.Marshal).

**Замечание**. В главе 3 уже отмечалось, что не допускается переопредешть Finalize() с типами структуры. Теперь это совершенно ясно, поскольку структуры являются типами, характеризуемыми значениями, и они не размещаются в динамической памяти.

**Переопределение System.Object.Finalize()**

В тех редких случаях, когда воздается C#-класс, использующий неуправляемые ресурсы, нужно гарантировать, что соответствующая память будет обрабатываться прогнозируемым образом. Предположим, что вы создали класс MyResourceWrapper, использующий неуправляемый ресурс (каким бы этот ресурс ни был), и вы хотите переопределить Finalize(). Немного странно, но в C# вы не можете для этого использовать ключевое слово override.

public class MyResourceWrapper {

 // Ошибка компиляции!

 protected override void Finalize(){}

}

Чтобы в пользовательском типе класса переопределить метод Finalize(), вы должны использовать в C# другой синтаксис деструктора, напоминающий синтаксис деструктора в C++. Причина в том, что при обработке деструктора компилятором C# в метод Finalize() автоматически добавляются необходимые элементы инфраструктуры (что будет продемонстрировано чуть позже).

Вот пример пользовательского деструктора для MyResourceWrapper, который при вызове генерирует системный сигнал. Ясно, что это сделано исключительно для примера. Настоящий деструктор должен только освобождать неуправляемые ресурсы, а не взаимодействовать с членами других управляемых o6ъeктов, поскольку вы не можете гарантировать, что эти объекты будут существовать в тот момент, когда сборщик мусора вызовет ваш метод Finalize().

**// Переопределение System.Object.Finalize() с использованием**

**// синтаксиса деструктора.**

class MyResourceWrapper {

 ~MyResourceWrapper() {

**// Освобождение неуправляемых ресурсов.**

**// Завершающий сигнал (только для примера!)**

  Console.Веер();

 }

}

Если рассмотреть этот деструктор с помощью ildasm.exe, вы увидите, что компилятор добавляет программный код контроля ошибок. Программный код вашего метода Finаlize() помещается в рамки блока try. Это делается для выявления операторов, которые во время выполнения могут сгенерировать ошибку (что формально называется *исключительной ситуацией*или *исключением).*Соответствующий блок finally гарантирует, что метод Finalize() класса будет выполнен независимо от исключений, которые могут возникать в рамках try. Формальности структурированной обработки исключений будут рассмотрены в следующей главе, а пока что проанализируйте следующее CIL-представление деструктора для C#-класса MyResourceWrapper.

.method family hidebysig virtual instance void Finalize() cil managed {

**// Code size 13 (0xd)**

 .maxstack 1

**.try**{

  IL\_0000: ldc.i4 0x4e20

  IL\_0005: ldc.i4 0x3e8

  IL 000a: call **void [mscorlib]System.Console::Beep(int32, int32)**

  IL\_000f: nop

  IL\_0010: nop

  IL\_0011: leave.s IL\_001b

 } // end.try

**finally**{

  IL\_0013: ldarg.0

**IL\_0014: call instance void [mscorlib]System.Object::Finalize**()

  IL\_0019: nop

  IL\_001a: endfinally

 } // end handler

 IL\_001b: nop

 IL\_001c: ret

} // end of method MyResourceWrapper::Finalize

При тестировании типа MyResourceWrapper вы обнаружите, что завершение работы приложения сопровождается системным звуковым сигналом, поскольку среда CLR автоматически вызывает деструкторы объектов при освобождении доменов приложений.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с деструкторами \*\*\*\*\*\n");

 Console.WriteLine("Нажмите клавишу ввода для завершения работы");

 Console.WriteLine("и вызова Finalize() сборщиком мусора");

 Console.WriteLine("для объектов, предусматривающих финализацию.");

 Console.ReadLine();

 MyResourceWrapper rw = new MyResourceWrapper();

}

**Исходный код.**Проект SimpleFinalize размещен в подкаталоге, соответствующем главе 5.

**Детали процесса финализации**

Чтобы не делать лишней работы, следует помнить о том, что целью метода Finalize() является гарантия освобождения неуправляемых ресурсов .NET-объекта при сборке мусора. Поэтому при cоздании типа, не использующего неуправляемые элементы (а такая ситуация оказывается вполне типичней), от финализации будет мало пользы. На самом деле, при разработке своих типов вы должны избегать использования метода Finalize() по той очень простой причине, что финализация требует времени.

При размещений объекта в управляемой динамической памяти среда выполнения автоматически определяет, поддерживает ли этот объект пользовательский метод Finalize(). Если указанный метод поддерживается, то объект обозначается, как *требующий финализации*, а указатель на этот объект сохраняется во внутренней очереди, которую называют *очередью финализации.*Очередь финализации представляет собой таблицу, поддерживаемую сборщиком мусора и содержащую все объекты, для которых перед удалением из динамической памяти требуется финализация.

Когда с точки зрения сборщика мусора приходит время удалить объект из памяти, проверяются элементы очереди финализации, и соответствующий объект копируется из динамической памяти в другую управляемую структуру, которую вызывают таблицей *элементов, доступных для финализации.*В этот момент создается отдельный поток, задачей которого является вызов метода Finalize() при *следующей сборке мусора*для каждого объекта из таблицы элементов, доступных для финализации. С учетом этого становится. ясно, что для окончательного уничтожения объекта потребуется как минимум *две*процедуры сборки мусора.

Таким образом, хотя финализация объекта и гарантирует, что объект сможет освободить неуправляемые ресурсы, по своей природе эта процедура оказывается недетерминированной и, в связи с происходящими "за кулисами" дополнительными процессами, достаточно медленной.

**Создание объектов, предусматривающих освобождение ресурсов**

Поскольку многие неуправляемые ресурсы являются столь "драгоценными", что их нужно освободить как можно быстрее, предлагается рассмотреть еще один подход, используемый для "уборки" ресурсов объекта. В качестве альтернативы переопределению Finalize() класс может реализовать интерфейс IDisposable, который определяет единственный метод, имеющий имя Dispose().

public interface IDisposable {

 void Dispose();

}

Если вы не имеете опыта программирования интерфейсов, то в главе 7 вы найдете все необходимые подробности. В сущности, интерфейс представляет собой набор абстрактных членов, которые могут поддерживаться классом или структурой. Если вы реализуете поддержку интерфейса IDisposable, то предполагается, что после завершения работы с объектом *пользователь объекта*должен вручную вызвать Dispose() до того, как объектная ссылка "уйдет" из области видимости. При таком подходе ваши объекты смогут выполнить всю необходимую "уборку" неуправляемых ресурсов без размещения в очереди финализации и без ожидания сборщика мусора, запускающего программную логику финализации класса.

**Замечание.**Интерфейс IDisposable может поддерживаться и типами структуры, и типами класса (в отличие от переопределения Finalize(), которое годится только для типов класса).

Ниже показан обновленный класс MyResourceWrapper, который теперь реализует IDisposable вместо переопределения System.Object.Finalize ().

// **Реализация IDisposable.**

public class MyResourceWrapper: **IDisposable**{

**// Пользователь объекта должен вызвать этот метод**

**// перед завершением работы с объектом.**

 public void Dispose() {

**// Освобождение неуправляемых ресурсов.**

**// Освобождение других содержащихся объектов.**

 }

}

Обратите внимание на то, что метод Dispose() отвечает не только за освобождение неуправляемых ресурсов типа, но и за вызов Dispose() для всех других содержащихся в его распоряжении объектов, предполагающих освобождение ресурсов. В отличие от Finalize(), обращаться из метода Dispose() к другим управляемым объектам вполне безопасно. Причина в том. что сборщик мусора не имеет никакого представления об интерфейсе IDisposable и никогда не вызывает Dispose(). Поэтому, когда пользователь объекта вызывает указанный метод, объект все еще существует в управляемой динамической памяти и имеет доступ ко всем другим объектам, размещенным в динамической памяти. Логика вызова проста.

public class Program {

 static void Main() {

  MyResourceWrapper rw = new MyResourceWrapper();

  rw.Dispose();

  Console.ReadLine();

 }

}

Конечно, перед попыткой вызвать Dispose() для объекта вы должны проверить, что соответствующий тип поддерживает интерфейс IDisposable. Обычно информацию об этом вы будете получать из документации .NET Framework 2.0 SDK, но это можно выяснить и программными средствами, используя ключевые слова is или as, применение которых обсуждалось в главе 4.

public class Program {

 static void Main() {

  MyResourceWrapper rw = new MyResourceWrapper();

  if (rw is IDisposable) rw.Dispose();

  Console.ReadLine();

 }

}

Этот пример заставляет вспомнить еще одно правило работы с типами, предполагающими сборку мусора.

*• Правило.*Обязательно вызывайте Dispose() для любого возданного вами объекта, поддерживающего IDisposable. Если разработчик класса решил реализовать поддержку метода Dispose(), то типу, скорее всего, есть что "убирать".

**Снова о ключевом слове using в C#**

При обработке управляемых объектов, реализующих интерфейс IDisposable, вполне типичным будет использование методов структурированной обработки исключений (см. главу 6), чтобы гарантировать вызов метода Dispose() даже при возникновении исключительных ситуаций в среде выполнения.

static void Main(string[] args) {

 MyResourceWrapper rw = new MyResourceWrapper();

 try {

**// Использование членов rw.**

 } finally {

**// Dispose () вызывается всегда, есть ошибки или нет.**

  rw.Dispose();

 }

}

Этот пример применения технологии "Безопасного программирования" прекрасен, но реальность такова, что лишь немногие разработчики готовы мириться с перспективой помещения каждого типа, предполагающего освобождение ресурсов, в рамки блока try/catch/finally только для того, чтобы гарантировать вызов метода Dispose(). Поэтому для достижения того же результата в C# предусмотрен намного более удобный синтаксис, реализуемый с помощью ключевого слова using.

static void Main(string[] args) {

**// Dispose() вызывается автоматически при выходе за пределы**

**// области видимости using.**

 using(MyResourceWrapper rw = new MyResourceWrapper()) {

**// Использование объекта rw.**

 }

}

Если с помощью ildasm.exe взглянуть на CIL-код метода Main(), то вы обнаружите, что синтаксис using на самом деле разворачивается в логику try/finally с ожидаемым вызовом Dispose().

.method private hidebysig static void Main(string [] args) cil managed {

 …

 .try {

  …

 } // end try

 finally {

  …

  IL\_0012: callvirt instance void **SimpleFinalize.MyResourceWrapper::Dispose**()

 } // end handler

} // end of method Program::Main

**Замечание.**При попытке применить using к объекту, не реализующему интерфейс IDisposable, вы получите ошибку компиляции.

Этот синтаксис исключает необходимость применения "ручной укладки" объектов в рамки программной логики try/finally, но, к сожалению, ключевое слово using в C# является двусмысленным (оно используется для указания пространств имен и для вызова метода Dispose()). Тем не менее, для типов .NET, предлагающих интерфейс IDisposable, синтаксическая конструкция using гарантирует автоматический вызов метода Dispose() при выходе из соответствующего блока.

**Исходный код.**Проект SimpleDispose размещен в подкаталоге, соответствующем главе 5.

**Создание типов, предусматривающих освобождение ресурсов и финализацию**

К этому моменту мы с вами обсудили два различных подхода в построении классов, способных освобождать свои внутренние неуправляемые ресурсы. С одной стороны, можно переопределить System.Object.Finalize(), тогда вы будете уверены в том, что объект непременно освободит ресурсы при сборке мусора, без какого бы то ни было вмешательства пользователя. С другой стороны, можно реализовать IDisposable, что обеспечит пользователю возможность освободить ресурсы после завершения работы с объектом. Однако, если вызывающая сторона "забудет" вызвать Dispose(), неуправляемые ресурсы смогут оставаться в памяти неопределенно долгое время.

Вы можете догадываться, что есть возможность комбинировать оба эти подхода в одном определении класса. Такая комбинации позволит использовать преимущества обеих моделей. Если пользователь объекта *не забудет* вызвать Dispose(), то с помощью вызова GC.SuppressFinalize() вы можете информировать сборщик мусора о том. что процесс финализации следует отменить. Еcли пользователь объекта*забудет*вызвать Dispose(), то объект, в конечном счете, подвергнется процедуре финализации при сборке мусора. Так или иначе, внутренние неуправляемые ресурсы объекта будут освобождены. Ниже предлагается очередной вариант MyResourceWrapper, в котором теперь предусмотрены и финализация, и освобождение ресурсов.

**// Сложный контейнер ресурсов.**

public class MyResourceWrapper: IDisposable {

**// Сборщик мусора вызывает этот метод в том случае, когда**

**// пользователь объекта забывает вызвать Dispose().**

 ~MyResourceWrapper() {

**// Освобождение внутренних неуправляемых ресурсов.**

**// НЕ следует вызывать Dispose() для управляемых объектов.**

 }

**// Пользователь объекта вызывает этот метод для того, чтобы**

**// как можно быстрее освободить ресурсы.**

 public void Dispose() {

**// Освобождение неуправляемых ресурсов.**

**// Вызов Dispose() для содержащихся объектов,**

**// предусматривающих освобождение ресурсов.**

**// Если пользователь вызвал Dispose(), то финализация не нужна.**

  GC.SuppressFinalize(this);

 }

}

Обратите внимание на то, что в метод Dispose() здесь добавлен вызов GC.SuppressFinalize(), информирующий среду CLR о том, что теперь при сборке мусора не требуется вызывать деструктор, поскольку неуправляемые ресурсы уже освобождены с помощью программной логики Dispose().

**Формализованный шаблон освобождения ресурсов**

Текущая реализация MyResourceWrapper работает вполне приемлемо, но некоторые недостатки она все же имеет. Во-первых, каждому из методов Finalize() и Dispose() приходится очищать одни и те же неуправляемые ресурсы. Это, конечно, ведет к дублированию программного кода, что усложняет задачу его поддержки. Лучше всего определить приватную вспомогательную функцию, которая вызывалась бы каждым из двух этих методов.

Далее, следует убедиться в том, что метод Finalize() не пытается освободить управляемые объекты, которые должны обрабатываться методом Dispose(). Наконец, желательно гарантировать, что пользователь объекта сможет многократно вызывать метод Disposed без появления сообщений об ошибке. В настоящий момент наш метод Dispose() таких гарантий не дает.

Для решения указанных проблемы Microsoft определила формализованный шаблон для освобождения ресурсов, устанавливающий баланс между устойчивостью к ошибкам, производительностью и простотой поддержки. Вот окончательная (и аннотированная) версия MyResourceWrapper, для которой используется этот "официальный" шаблон.

public class MyResourceWrapper: IDisposable {

**// Используется для того, чтобы выяснить,**

**// вызывался ли метод Dispose().**

 private bool disposed = false;

 public void Dispose() {

**// Вызов нашего вспомогательного метода.**

**// Значение "true" указывает на то, что**

**// очистку инициировал пользователь объекта.**

  CleanUp(true);

  // **Запрет финализации.**

  GC.SuppressFinalize(this);

 }

 private void CleanUp(bool disposing) {

  // **Убедимся, что ресурсы еще не освобождены.**

  if (!this.disposed) {

**// Если disposing равно true, освободить**

**// все управляемые ресурсы.**

   if (disposing) {

**// Освобождение управляемых ресурсов.**

   }

**// Освобождение неуправляемых ресурсов**.

  }

  disposed = true;

 }

 ~MyResourceWrapper() {

**// Вызов нашего вспомогательного метода.**

**// Значение "false" указывает на то, что**

**// очистку инициировал сборщик мусора.**

  CleanDp(false);

 }

}

Обратите внимание на то, что теперь MyResourceWrapper определяет приватный вспомогательный метод, с именем Cleanup(). Если для его аргумента указано true (истина), это значит, что сборку мусора инициировал пользователь объекта. И тогда мы должны освободить и управляемые, и неуправляемые ресурсы. Но если "уборка" инициирована сборщиком мусора, то при вызове CleanUp() следует указать false (ложь), чтобы внутренние объекты не освобождались (поскольку мы не можем гарантировать, что они все еще находятся в памяти). Наконец, перед выходом из CleanUp() член-переменная disposed логического типа устанавливается равной true, чтобы Dispose() можно было вызывать многократно без появления сообщений об ошибках.

**Исходный код.**Проект FinalizableDisposableClass размещен в подкаталоге, соответствующем главе 5.

На этом наше обсуждение того, как среда CLR управляет объектами с помощью сборки мусора, завершается. Остались нерассмотренными еще ряд вопросов, связанных с процессом сборки мусора (это, например, слабые ссылки и восстановление объектов), но вы теперь имеете достаточно знаний для того, чтобы продолжить исследование данной темы самостоятельно в удобное для вас время.

**Резюме**

Целью этой главы было обсуждение процесса сборки мусора. Вы могли убедиться, что сборщик мусора начинает работу только тогда, когда становится невозможным выделить необходимый объем управляемой динамической памяти (или когда из памяти выгружается домен данного приложения). Вы можете быть уверены в том, что сборка мусора будет выполнена в оптимальном режиме с помощью соответствующего алгоритма Microsoft, использующего генерации объектов, вторичные потоки для финализации, объектов и области управляемой динамической памяти, предназначенные для размещения больших объектов.

В этой же главе объясняется, как с помощью типа класса System.GC взаимодействовать со сборщиком мусора. Это требуется при создании типов, предусматривающих финализацию или освобождение ресурсов. Типы, предусматривающие финализацию, представляют собой классы с переопределенным виртуальным методом System.Object.Finalize(), который (в будущем) должен обеспечить освобождение неуправляемых ресурсов. Объекты, предуcматриваюцие освобождение ресурсов, являются классами (или структурами), в которых реализуется интерфейс IDisposable. В рамках этого подхода пользователю объекта предлагается открытый метод, который должен быть вызван пользователем для выполнения внутренней ''уборки" сразу же, как только это потребуется. Наконец, вы узнали об "официальном" формализованном шаблоне освобождения ресурсов, в котором комбинируются оба указанных подхода.

**ГЛАВА 6. Структурированная обработка исключений**

Тема этой главы – устранение аномалий в среде выполнения вашего программного кода C# с помощью *структурированной обработки исключений*. Вы узнаете о ключевых словах C#, которые позволяют решать такие задачи (это ключевые слова try, catch, throw, finally), и выясните, в чем различие между исключениями системного уровня и уровня приложений. Данное обсуждение можно рассматривать, как введение в тему создания пользовательских исключений, а также как краткое описание средств отладки Visual Studio 2005, в основе которых, по сути, и лежит обработка исключений.

**Ода ошибкам и исключениям**

Вне зависимости от того, что говорит нам наша (часто преувеличенная) самооценка, абсолютно идеальных программистов не существует. Создание программного обеспечения является сложным делом и поэтому даже самые лучшие программы могут иметь *проблемы.*Иногда причиной проблем бывает несовершенный: программный код (например, выход за границы массива). В других случаях проблема возникает из-за неправильных данных, вводимых пользователем (например, ввод имени в поле телефонного номера), если возможность обработки таких данных не была предусмотрена в приложении. Независимо от причин возникающих проблем, результатом всегда оказывается непредусмотренное поведение приложения. Чтобы очертить рамки нашего обсуждения структурированной, обработки исключительных ситуаций, позвольте сформулировать определения трех терминов, часто используемых при изучении соответствующих аномалий.

• *Программные ошибки.*Это, попросту говоря, ошибки программиста. Например, при использовании языка C++ без управляемых расширений, если вызвать указатель NULL или забыть очистить выделенную память (в результате чего происходит ''утечка" памяти), то возникает программная ошибка.

• *Пользовательские ошибки.*В отличие от программных ошибок, пользовательские ошибки обычно возникают в результате действий объекта, использующего приложение, а не в результате действий разработчика. Например, действия пользователя, который вводит в текстовый блок строку "неправильного вида", могут привести к возникновению ошибки, *если*ваш программный код не в состоянии обработать такой ввод.

• *Исключения*. Исключения, или исключительные ситуации, обычно проявляются в виде аномалий среды выполнения, и очень трудно, если вообще возможно, полностью учесть их все в процессе программирования приложения. К возможным исключительным ситуациям относятся попытки соединиться с базой данных, которой больше не существует, открыть поврежденный файл, контактировать с компьютером, который выключен. Программист, как и конечные пользователи, имеют очень ограниченные возможности управления в подобных "исключительных обстоятельствах".

С учетом данных определений должно быть ясно, что структурированная обработка исключений в .NET предлагает подход, предназначенный для выявления исключительных ситуаций в среде выполнения. Однако и в случае программных и пользовательских ошибок, которые ускользнули от вашего внимания, среда CLR зачастую генерирует соответствующее исключение, идентифицирующее проблему. Для этого библиотеки базовых классов .NET определяют целый ряд исключений, таких как FormatException, IndexOutOfRangeException, FileNotFoundException, ArgumentOutOfRangeExсeption и т.д.

Перед тем как углубиться в детали, давайте определим формальную роль структурированной обработки исключений и выясним, чем такая обработка отличается от традиционного подхода в деле исправления ошибок.

**Замечание.**Чтобы сделать примеры программного кода, используемые в данной книге, как можно более конкретными, в них обрабатываются не асе возможные исключения, генерируемые соответствующими методами из библиотек базовых классов. Приемы, представленные в этой главе, вы можете использовать в своих реальных проектах по своему усмотрению.

**Роль обработки исключений в .NET**

До появления .NET обработка ошибок в операционной системе Windows представляла собой весьма запутанную смесь технологий. Многие программисты создавали свою собственную логику обработки ошибок, используемую в контексте приложения. Например, группа разработчиков могла определить набор числовых констант, представляющих известные сбойные ситуации, и использовать эти константы в качестве возвращаемых значений метода. Для примера взгляните на следующий фрагмент программного кода, созданного на языке C.

**/\* Типичный механизм учета ошибок в C. \*/**

#define E\_FILENOTFOUND 1000

int SomeFunction() {

**// Предположим, что возникла ситуация, в результате**

**// которой возвращается следующее значение.**

 return E\_FILENOTFOUND;

}

void Main() {

 int retVal = SomeFunction();

 if (retVal == E\_FILE\_NOTFOUND) printf("Не найден файл…");

}

Такой подход следует признать далеким от идеального, если учесть тот факт, что константа E\_FILENOTFOUND является лишь числовым значением, а не информационным агентом, предлагающим решение возникшей проблемы. В идеале хотелось бы получать название ошибки, сообщение о ней и другую полезную информацию в одном пакете вполне определенного вида (и именно это предлагается в рамках структурированного подхода к обработке исключений).

В дополнение к приемам самого разработчика, Windows API предлагает сотни кодов ошибок, которые поставляются в виде #define, HRESULT, а также в виде многочисленных вариаций булевых значений (bool, BOOL, VARIANT\_BOOL и т.д.). Многие разработчики программ на языке C++ (а также VB6) в рамках модели COM явно или неявно применяют ограниченный набор стандартных COM-интерфейсов (например, ISupportErrorInfo, IErrorInfo, ICreateErrorInfo), чтобы предоставить COM-клиенту информацию об ошибках.

Очевидной проблемой этой уже устаревшей схемы является отсутствие симметрии. Каждый из подходов более или менее укладывается в рамки своей конкретной технологии, конкретного языка и, возможно, даже в рамки конкретного проекта. Чтобы положить конец неуемному буйству разнообразия, платформа .NET предлагает стандартную технологию генерирования и выявления ошибок среды выполнения: структурированную обработку исключений – СОИ (Structured Exception Handling – SEH).

Преимущество предлагаемой технологии заключается в том. что теперь разработчики могут использовать в области обработки ошибок унифицированный подход, общий для всех языков, обеспечивающих поддержку .NET. Таким образом, способ обработки ошибок в C# оказывается синтакcически аналогичным такой обработке и в VB .NET, и в C++ с управляемыми расширениями. При этом синтаксис, используемый для генерирования и выявления исключений между компоновочными блоками и границами систем, тоже оказывается одинаковым (и это является дополнительным преимуществом).

Еще одним преимуществом обработки исключений в .NET является то, что вместо передачи простого "зашифрованного" числового значения, идентифицирующего проблему, исключения представляют собой объекты, содержащие понятное человеку описание ошибки, а также подробную "копию" содержимого стека вызовов в момент возникновений исключительной ситуации. К тому же вы имеете возможность предоставить конечному пользователю ссылку с адресом URL, по которой пользователь может получить подробную информацию о соответствующей проблеме.

**Атомы обработки исключений в .NET**

При создании программ с применением структурированной обработки исключений предполагается использовать следующие четыре взаимосвязанных элемента:

• тип класса, который предоставляет подробную информацию о возникшей исключительной ситуации;

• член, который *генерирует,*или *направляет*(throw) вызывающей стороне экземпляр класса, соответствующего исключительной ситуации:

• блок программного кода вызывающей стороны, в котором был вызван генерирующий исключение член;

• блок программного кода вызывающей стороны, в котором выполняется *обработка,*или *захват*(catch), данного исключения.

В языке программирования C# предлагаются четыре ключевых слова (try, catch, throw и finally), с помощью которых генерируются и обрабатываются исключения. Тип, представляющий соответствующую проблему, является классом, производным от System.Exception (или его потомком). С учетом этого давайте выясним роль указанного базового класса.

**Базовый класс System.Exception**

Все исключения, определенные на уровне пользователя и системы, в конечном счете получаются из базового класса System.Exception (который, в свою очередь, получается из System.Object). Обратите внимание на то, что некоторые из указанных ниже членов виртуальны и поэтому могут переопределяться производными типами.

public class **Exception:** ISerializable, \_Exception {

 public virtual IDictionary Data { get; }

 protected Exception(SerializationInfo info, StreamingContext context);

 public Exception(string message, Exception innerException);

 public Exception(string message);

 public Exception();

 public virtual Exception GetBaseException();

 public virtual void GetObjectData(SerializationInfo info, StreamingContext context);

 public System.Type GetType();

 protected int HResult { get; set; }

 public virtual string HelpLink { get; set; }

 public System.Exception InnerException { get; }

 public virtual string Message { get; }

 public virtual string Source { get; set; }

 public virtual string StackTrace { get; }

 public MethodBase TargetSite { get; }

 public override string ToString();

}

Как видите, многие свойства, определенные в классе System.Exception, доступны только для чтения. Причиной этого является тот простой факт, что производные типы обычно предусматривают для каждого свойства значение по умолчанию (например, для типа IndexOutOfRangeException принятым по умолчанию сообщением является "Index was outside the bounds of the array", т.е. "Выход индекса за границы массива").

**Замечание**. В .NET 2.0 System.Exception реализует интерфейс \_Exception*,*чтобы соответствующие функциональные возможности были доступны неуправляемому программному коду.

В табл. 6.1 предлагаются описаний некоторых членов System.Exception.

**Таблица 6.1.** Основные члены типа System.Exception

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Data | Добавлено в .NET 2.0. Предлагает коллекцию пар ключей и значений (пред-cтавленную объектом, реализующим IDictionary), которая обеспечивает дополнительную пользовательскую информацию о данном исключении. По умолчанию эта коллекция является пустой |
| HelpLink | Возвращает адрес URL файла справки с описанием ошибки во всех подробностях |
| InnerException | Доступно только для чтения. Может использоваться для получения информации о предыдущем исключении или исключениях, ставших причиной данного исключения. Запись предыдущих Исключений осуществляется путем передачи их конструктору самого последнего исключения |
| Message | Доступно только для чтения. Возвращает текстовое описание данной ошибки. Само сообщение об ошибке задается, как параметр конструктора |
| Source | Возвращает имя компоновочного блока, генерирующего исключение |
| StackTrace | Доступно только для чтения. Содержит строку, идентифицирующую последовательность вызовов, которые привели к исключительной ситуации. Как вы можете догадаться сами, это свойство оказывается очень полезным для отладки |
| TargetSite | Доступно только для чтения. Возвращает тип MethodBase, предлагающий самую разную информацию о методе, который генерировал исключение (ToString() будет идентифицировать имя соответствующего метода) |

**Простейший пример**

Чтобы продемонстрировать "пользу" структурированной обработки исключений, нужно создать тип, который в подходящем окружении может генерировать исключение. Предположим, что мы создали новое консольное приложение с именeм SimpleException, в котором определяются два типа класса Car (автомобиль) и Radio (радио), связанные отношением локализации ("has-a"). Тип Radio определяет один метод, включающий и выключающий радио.

public class Radio {

 public void TurnOn(bool on) {

  if (on) Console.WriteLine("Радиопомехи…");

  else Console.WriteLine ("И тишина…");

 }

}

В дополнение к использованию типа Radio в рамках модели локализации/делегирования, тип Car определяет следующее поведение. Если пользователь объекта Car превысит предел для скорости (этот предел задается значением соответствующего члена-константы), то "двигатель взрывается" и объект Car становится непригодным для использования, что выражается в соответствующем изменении значения члена-переменной типа bool с именем carIsDead (автомобиль разрушен). Кроме того, тип Car имеет несколько членов-переменных, представляющих текущую скорость и "ласкательное имя", данное автомобилю пользователем, а также несколько конструкторов. Вот полное определение этого типа (с соответствующими примечаниями).

public class Car {

**// Константа для максимума скорости.**

 public const int maxSpeed = 100;

 // **Данные внутреннего состояния.**

 private int currSpeed;

 private string petName;

 // **Работает ли этот автомобиль?**

 private bool carIsDead;

**// В автомобиле есть радио.**

 private Radio theMusicBox = new Radio();

 // **Конструкторы.**

 public Car() {}

 public Car(string name, int currSp) {

  currSpeed = currSp;

  petName = name;

 }

 public void CrankTunes(bool state) {

  // **Запрос делегата для внутреннего объекта.**

  theMusicBox.TurnOn(state);

 } // **He перегрелся ли автомобиль?**

 public void Accelerate(int delta) {

  if (carIsDead) Console.WriteLine("{0} не работает…", petName);

  else {

   currSpeed += delta;

   if (currSpeed › maxSpeed) {

    Console.WriteLine("{0} перегрелся!", petName);

    currSpeed = 0; carIsDead = true;

   } else Console.WriteLine("=› currSpeed = {0}", currSpeed);

  }

 }

}

Теперь реализуем такой метод Main(), в котором объект Car превысит заданную максимальную скорость (представленную) константой maxSpeed).

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Создание и испытание автомобиля \*\*\*");

 Car myCar = new Car("Zippy", 20);

 myCar.CrankTunes(true);

 for (int i = 0; i ‹ 10; i++) myCar.Accelerate(10);

 Console.ReadLine();

}

Тогда мы увидим вывод, подобный показанному на рис. 6.1.

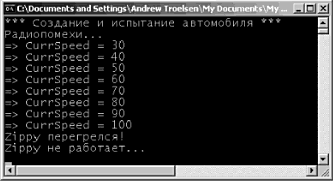


Рис. 6.1. Объект Car в действии

**Генерирование исключений**

Теперь, когда тип Car функционирует, продемонстрируем простейший способ генерирования исключений. Текущая реализация Accelerate() выводит сообщение об ошибке, когда значение скорости объекта Car становится выше заданного предела.

Модифицируем этот метод так, чтобы он генерировал исключение при попытке пользователя увеличить скорость автомобиля выше предусмотренных его создателем пределов. Для этого нужно создать и сконфигурировать экземпляр класса System.Exception, установив значение доступного только для чтения свойства Message с помощью конструктора класса. Чтобы отправить соответствующий ошибке объект вызывающей стороне, используйте ключевое слово C# throw. Вот как может выглядеть соответствующая модификация метода Accelerate().

**// Теперь, если пользователь увеличит скорость выше maxSpeed,**

**// генерируется исключение.**

public void Accelerate(int delta) {

 if (carIsDead) Console.WriteLine("{0} не работает…", petName);

 else {

  currSpeed += delta;

  if (currSpeed ›= maxSpeed) {

   carIsDead = true;

   currSpeed = 0;

**// Используйте ключевое слово "throw",**

**// чтобы генерировать исключение.**

   throw new Exception(string.Format("{0} перегрелся!", petName));

  } else Console.WriteLine("=› CurrSpeed = {0}', currSpeed);

 }

}

Прежде чем выяснить, как вызывающая сторона должна обрабатывать это исключение, отметим несколько интересных моментов. Во-первых, при генерировании исключения только от вас зависит, что следует считать исключительной ситуацией и когда должно быть сгенерировано соответствующее исключение. Здесь мы предполагаем, что при попытке в программе увеличить скорость автомобиля, который уже перестал функционировать, должен генерироваться тип System.Exception, сообщающий, что метод Accelerate() не может продолжить свою работу.

В качестве альтернативы можно реализовать такой метод Accelerate(), который автоматически выполнит восстановление без генерирования исключения. Вообще говоря, исключения должны генерироваться только тогда, когда выявляются экстремальные ситуации (например, не найден необходимый файл, нет возможности соединиться с базой данных и т.п.). Решение о том, что именно должно вызывать появление исключений, должно приниматься разработчиком, и вы, как разработчик, должны всегда помнить об этом. Для наших примеров мы предполагаем, что увеличение скорости нашего обреченного автомобиля выше допустимого предела является вполне серьезной причиной для генерирования исключения.

**Обработка исключений**

Ввиду того, что теперь метод Accelerate() может генерировать исключение, вызывающая сторона должна быть готова обработать такое исключение. При вызове метода, способного генерировать исключение, вы должны использовать блок try/catch. Приняв исключение, вы можете вызвать члены типа System.Exception и прочитать подробную информацию о проблеме. Что вы будете делать с полученными данными, зависит, в основном, от вас. Вы можете поместить соответствующую информацию в файл отчета, записать ее в журнал регистрации событий Windows, отправить ее по электронной почте администратору системы или показать сообщение с описанием проблемы конечному пользователю. Здесь мы просто выводим информацию в окно консоли.

**// Обработка сгенерированного исключения.**

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Создание и испытание автомобиля \*\*\*");

 Car myCar = new Car("Zippy", 20);

 myCar.CrankTunes(true);

**// Превышение допустимого максимума для скорости,**

**// чтобы генерировать исключение.**

 try {

  for (int i = 0; i ‹ 10; i++) myCar.Accelerate(10);

 } catch(Exception e) {

  Console.WriteLine("\n*\*\**\* Ошибка! *\*\*\**");

  Console.WriteLine("Метод: {0}", e.TargetSite);

  Console.WriteLine("Сообщение: {0}", e.Message);

  Console.WriteLine("Источник: {0}", е.Source);

 }

**// Ошибка обработана, выполняется следующий оператор.**

 Console.WriteLine("\n\*\*\* Выход из обработчика исключений *\*\*\**");

 Console.ReadLine();

}

Блок try представляет собой набор операторов, которые могут генерировать иcключения в ходе их выполнения. Если обнаруживается исключение, поток выполнения программы направляется подходящему блоку catch. С другой стороны, если программный код в рамках блока try не генерирует исключений, блок catch полностью пропускается, и все проходит "тихо и спокойно". На рис. 6.2 показан вывод этой программы.

Как видите, после обработки исключения приложение может продолжать свою работу, начиная с оператора, следующего за блоком catch. В некоторых случаях исключение может оказаться достаточно серьезным для того, чтобы прекратить выполнение приложения. Но в большинстве случаев оказывается возможным исправить ситуацию в рамках программной логики обработчика исключений, чтобы приложение продолжило свою работу (хотя и с возможными ограничениями функциональности, если, например, не удается установить соединение с удаленным источником данных).

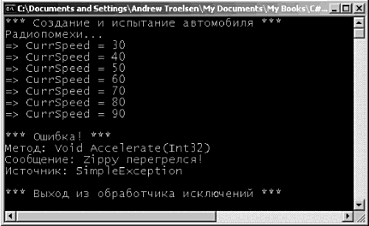


Рис. 6.2. Визуализация ошибок в рамках структурированной обработки исключений

**Конфигурация состояния исключений**

В настоящий момент конфигурация нашего объекта System.Exception задается в методе Accelerate(), где устанавливается значение, приписываемое свойству Message (через параметр конструктора). Но, как следует из табл. 6.1, класс Exception предлагает ряд дополнительных членов (TargetSite, StackTrace, HelpLink и Data), которые могут оказаться полезными в процессе дальнейшего анализа возникшей проблемы. Чтобы усовершенствовать наш пример, давайте рассмотрим содержимое указанных членов.

**Свойство TargetSite**

Свойство System.Exception.TargetSite позволяет выяснить дополнительную информацию о методе, генерирующем данное исключение. Как показано в предыдущем варианте метода Main(), при выводе значения TargetSite демонстрируется возвращаемое значение, имя и параметры метода, генерирующего данное исключение. Но TargetSite возвращает не просто строку, а строго типизированный объект System.Reflection.MethodBase. Этот тип содержит подробную информацию о методе, породившем проблему, и том классе, который определяет данный метод. Для иллюстрации обновим предыдущую логику catch так, как показано ниже.

static void Main(string[] args) {

 …

**// В действительности TargetSite возвращает объект MethodBase.**

 catch(Exception e) {

  Console.WriteLine("\n\*\*\* Ошибка! \*\*\*");

  Console.WriteLine("Имя члена: {0}", е.TargetSite);

  Console.WriteLine("Класс, определяющий метод: {0}", **е**.**TargetSite.DeclaringType);**

  Console.WriteLine("Тип члена: {0}", **е**.**TargetSite.MemberType**);

  Console.WriteLine("Сообщение: {0}", e.Message);

  Console.WriteLine("Источник: {0}", e.Source);

 }

 Console.WriteLine("\n\*\*\* Выход из обработчика исключений \*\*\*");

**myCar.Accelerate(10); // Это не ускорит автомобиль.**

Consolе.ReadLine();

}

На этот раз вы используете свойство MethodBase.DeclaringType, чтобы определить абсолютное имя класса, сгенерировавшего ошибку (в данном случае это класс SimpleException.Car), и свойство MemberType объекта MethodBase, чтобы идентифицировать тип породившего исключение члена (в том смысле, свойство это или метод). На рис. 6.3 показан обновленный вывод.

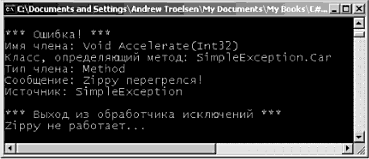


Рис 6.3. Получение информации о целевом объекте

**Свойство StackTrace**

Свойство System.Exception.StackTrace позволяет идентифицировать серию вызовов, которые привели к исключительной ситуации. Вы не должны устанавливать значение StackTrace, поскольку это делается автоматически в момент создания исключения. Для иллюстрации предположим, что мы снова обновили программный код catch.

catch (Exception e) {

 …

 Console.WriteLine(''Стек {0}", **e.StackTrace);**

}

Если выполнить программу теперь, то вы обнаружите, что на консоль будет выведен след стека (для вашего приложения номера строк и названия папок, конечно же, могут быть другими).

Стек: at SimpleException.Car.Accelerate(Int32 delta)

in с:\myаррs\exceptions\car.cs: line 65

at Exceptions.App.Main()

in с:\myapps\exceptions\app.cs: line 21

Строка, возвращаемая из StackTrace, сообщает последовательность вызовов, которые привели к генерированию данного исключения. Обратите внимание на то, что здесь последняя из возвращенных строк идентифицирует первый вызов в последовательности, а строка с наивысшим номером идентифицирует проблемный член. Эта информация может быть полезной при отладке приложения, поскольку вы получаете возможность "пройти" путь до источника ошибки.

**Свойство HelpLink**

Свойства Target Site и StackTrace позволяют получить информацию о данном исключении программисту, но конечному пользователю эта информация мало что дает. Вы уже видели, что для получения информации, понятной обычному пользователю, можно использовать свойство System.Exception.Message. В дополнение к этому свойство HelpLink может указать адрес URL или стандартный файл справки Windows, содержащий более подробную информацию.

По умолчанию значением свойства HelpLink является пустая строка. Чтобы присвоить этому свойству некоторое значение, вы должны сделать это перед тем, как будет сгенерирован тип System.Exception. Вот как можно соответствующим образом изменить метод Car.Accelerate().

public void Accelerate(int delta) {

 if (carIsDead) Console.WriteLine("{0) не работает…", petName);

 else {

  currSpeed += delta;

  if (currSpeed ›= maxSpeed) {

   carIsDead = true;

   currSpeed = 0;

**// Чтобы вызвать свойство HelpLink, перед оператором,**

**// генерирующим объект Exception, создается локальная переменная.**

   Exception ex = new Exception(string.Format("{0} перегрелся!", petName));

**ex.HelpLink = "http://www.CarsRUs.com";**

   throw ex;

  } else Console.WriteLine("=› CurrSpeed = {0}", currSpeed);

 }

}

Логику catch теперь можно изменить, чтобы информация ссылки выводилась так, как показано ниже.

catch(Exception e) {

 …

 Console.WriteLine("Соответствующая справка: {0}", **e.HelpLink);**

}

**Свойство Data**

Свойство Data объекта System.Exception является новым в .NET 2.0 и позволяет добавить в объект исключения дополнительную информацию для пользователя (например, штамп времени или что-то другое). Свойство Data возвращает объект, реализующий интерфейс с именем IDictionary, определенный в пространстве имен System.Collection. Роль программирования интерфейсов, как и пространство имен System.Collection, рассматриваются в следующей главе. Сейчас же будет достаточно заметить, что коллекции словарей позволяют создавать множества значений, возвращаемых по значению ключа. Рассмотрите, например, следующую модификацию метода Car.Accelerate().

public void Accelerate(int delta) {

 if (carIsDead) Console.WriteLine("{0} не работает…", petName);

 else {

  currSpeed += delta;

  if (currSpeed ›= maxSpeed) {

   carIsDead = true;

   currSpeed = 0;

**// Чтобы вызвать свойство HelpLink, перед оператором,**

**// генерирующим объект Exception, создается локальная переменная.**

   Exception ex = new Exception(string.Format("{0} перегрелся!", petName));

   ex.HelpLink = "http://www.CarsRUs.com";

**// Место для пользовательских данных с описанием ошибки.**

   ex.Data.Add("Дата и время", string.Format("Автомобиль сломался {0}", DateTime.Now));

   ex.Data.Add("Причина", " У вас тяжелая нога");

   throw ex;

  } else Console.WriteLine("=› CurrSpeed = {0}", currSpeed);

 }

}

Чтобы не возникло проблем при определении пар "ключ-значение", с помощью директивы using следует указать пространство имен System.Collection, поскольку в файле, содержащем класс с реализацией метода Main(), мы собираемся использовать тип DictionaryEntry.

using System.Collections;

Затем нужно обновить программную логику catch для проверки того, что значение, возвращаемое свойством Data, не равно null (значение null задается по умолчанию). После этого мы используем свойства Key и Value типа DictionaryEntry, чтобы вывести пользовательские данные на консоль.

catch (Exception e) {

 …

**// По умолчанию поле данных пусто, поэтому проверяем на null.**

 Console.WriteLine("\n-› Пользовательские данные:");

 if (e.Data != null) {

  foreach (DictionaryEntry de in e.Data) Console.WriteLine("-› {0}; {1}", de.Key, de.Value);

 }

}

С этими изменениями мы должны получить вывод, показанный на рис. 6.4.

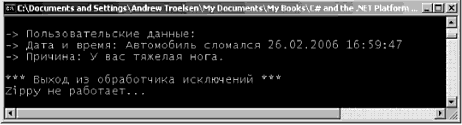


Рис. 6.4. Получение пользовательских данных

**Исходный код.**Проект SimpleException размещен в подкаталоге, соответствующем главе 6.

**Исключения системного уровня (System.SystemException)**

Библиотеки базовых классов .NET определяют множество классов, производных от System.Exception. Пространство имен System определяет базовые объекты ошибок, например ArgumentOutOfRangeException, IndexOutOfRangeException, StackOverflowException и т.д. Другие пространства имен определяют исключения, отражающие поведение своих элементов (например, System.Drawing.Printing определяет исключения, возникающие при печати, System.IO – исключения ввода-вывода, System.Data – исключения, связанные с базами данных и т.д.).

Исключения, генерируемые общеязыковой средой выполнения (CLR), называют исключениями *системного уровня.*Эти исключения считаются неустранимыми, фатальными ошибками. Исключения системного уровня получаются непосредственно из базового класса System.SystemException, являющегося производным от System.Exception (который, в свою очередь, получается из System.Object).

public class SystemException: Exception {

**// Различные конструкторы.**

}

С учетом того, что тип System.SystemException не добавляет ничего нового, кроме набора конструкторов, у вас может возникнуть вопрос, почему SystemException оказывается на первом месте. Главная причина в том, что если полученный тип исключения оказывается производным от System.SystemException, вы можете утверждать, что исключение сгенерировано средой выполнения .NET, а не программным кодом выполняемого приложения.

**Исключения уровня приложения (System.ApplicationException)**

Учитывая то, что все исключения .NET являются типами класса, можно создавать свои собственные исключения, учитывающие специфику приложения. Однако ввиду того, что базовый класс System.SystemException представляет исключения, генерируемые средой CLR, вполне естественно было бы предположить, что пользовательские исключений должны выводиться из типа System.Exception. Это действительно возможно, но практика диктует свой правила, по которым пользовательские исключения лучше выводить из типа System.ApplicationException.

public class ApplicationException: Exception {

**// Различные конструкторы.**

}

Подобно SystemException, тип ApplicationException не определяет никаких дополнительных членов, кроме набора конструкторов. С точки зрения функциональности единственной целью System.ApplicationException должна быть идентификация источника (устранимой) ошибки. При обработке исключения, полученного из System.ApplicationException, вы можете предполагать, что причиной появления исключения был программный код выполняемого приложения, а не библиотеки базовых классов .NET.

**Создание пользовательских исключений, раз…**

Всегда есть возможность генерировать экземпляр System.Exceptiоn, чтобы сигнализировать об ошибке времени выполнения (как показано в нашем первом примере), но часто бывает выгоднее построить*строго типизированное исключение,*которое предоставит уникальную информацию, характеризующую данную конкретную проблему. Предположим, например, что мы хотим создать пользовательское исключение (с именем CarIsDeadException), представляющее ошибку превышения скорости нашего обреченного автомобиля. Первым делом здесь должно быть создание нового класса из System.ApplicationException (по соглашению, классы исключений имеют суффикс "Exception", что в переводе означает "исключение").

**// Это пользовательское исключение предлагает описание**

**// автомобиля, вышедшего из строя.**

public class CarIsDeadException: ApplicationException {}

Как и в случае любого другого класса, можно определить пользовательские члены, которые будут затем использоваться в блоке catch в рамках программной логики вызовов. Точно так же можно переопределить любые виртуальные члены, определенные родительскими классами. Например, можно реализовать CarIsDeadException, переопределив виртуальное свойство Message.

public class CarIsDeadException: ApplicationException {

 private string messageDetails;

 public CarIsDeadException() {}

 public CarIsDeadException(string message) {

  messageDetails = message;

 }

**// Переопределение свойства Exception.Message.**

 public override string Message {

  get {

   return string.Format("Сообщение об ошибке Car: {0}", messageDetails);

  }

 }

}

Здесь тип CarIsDeadException предлагает приватный член (messageDetails), представляющий информацию о текущем исключении, которая может быть задана с помощью пользовательского конструктора. Генерировать ошибку с помощью Accelerate() очень просто. Здесь следует разместить, сконфигурировать и сгенерировать тип CarIsDeadException, а не общий тип System.Exception.

**// Генерируем пользовательское исключение CarIsDeadException.**

public void Accelerate(int delta) {

 ...

 CarIsDeadException ex = new CarIsDeadException(string.Format("{0} перегрелся!", petName));

**ex**.**HelpLink = "http://www.CarsRUs**.**com"**;

 ex.Data.Add("Дата и время", string.Format("Автомобиль сломался {0}", DateTime.Now));

 ex.Data.Add("Причина", "У вас тяжелая нога.");

 throw ex;

}

Чтобы выполнить явный захват поступившего исключения, блок catch следует изменить для захвата конкретного типа CarIsDeadException (однако, с учетом того, что System.CarIsDeadException является потомком System.Exception, можно также выполнить захват объекта System.Exception общего вида).

static void Main (string[] args) {

 catch (**CarIsDeadException** e) {

**// Обработка поступившего исключения.**

 }

}

Теперь, после рассмотрения основных этапов процесса создания пользовательских исключений, возникает вопрос: когда может потребоваться их создание? Как правило, пользовательские исключения требуется создавать только тогда, когда ошибка непосредственно связана с классом, генерирующим эту ошибку (например, пользовательский класс File может генерировать ряд исключений, связанных с доступом к файлам, класс Car может генерировать ряд исключений, связанных с работой автомобиля и т.д.). Тем самым вы обеспечиваете вызывающей стороне возможность обработки множества исключений на основе "индивидуального подхода".

**Создание пользовательских исключений, два…**

Тип CarIsDeadException переопределяет свойство System.Exception.Message, чтобы установить пользовательское сообщение об ошибке. Однако задачу можно упростить, установив родительское свойство Message через входной параметр конструктора. В результате нам не придется делать ничего, кроме следующего.

public class CarIsDeadException: ApplicationException {

 public CarIsDeadException() {}

 public CarIsDeadException(string message) : base (message) {}

}

Обратите внимание на то, что теперь мы не определяем строковую переменную для представления сообщения и не переопределяем свойство Message. Вместо этого конструктору базового класса просто передается параметр. В рамках такого подхода пользовательский класс исключений представляет собой немногим большее, чем класс с уникальным именем, полученный из System.ApplicationException без каких бы то ни было членов-переменных (и переопределений базового класса).

Не удивляйтесь, если большинство ваших пользовательских классов исключений (если не все они) будет иметь такой простой вид. Во многих случаях целью создания пользовательского исключения является не функциональные возможности, расширяющие возможности базовых классов, а получение *строго именованного типа,*идентифицирующего природу возникшей ошибки.

**Создание пользовательских исключений, три!**

Если вы хотите построить "педантично точный" пользовательский класс исключения, то созданный вами тип должен соответствовать лучшим образцам, использующим исключения .NET. В частности, ваше пользовательское исключение должно подчиняться следующим требованиям:

• быть производным от Exception/ApplicationException;

• обозначаться атрибутом [System.Serializable];

• определять конструктор, используемый по умолчанию;

• определять конструктор, устанавливающий наследуемое свойство Message;

• определять конструктор, обрабатывающий "внутренние исключения";

• определять конструктор, выполняющий сериализацию типа.

Пока что глубина ваших знаний .NET не позволяет вам понять роль атрибутов и сериализации объектов, но сейчас это и не важно. Соответствующие вопросы будут рассмотрены позже. А в завершение обзора, посвященного вопросам создания пользовательских исключений, рассмотрите заключительный вариант CarIsDeadException.

[Serializable]

public class CarIsDeadException: ApplicationException {

 public CarIsDeadException() {}

 public CarIsDeadException(string message): base (message) {}

 public CarIsDeadException(string message, System.Exception inner): base (message, inner) {}

 protected CarIsDeadException(System.Runtime.Serialization.SerializationInfо info, System.Runtime.Serialization.StreamingContext context) : base(info, context) {}

}

Пользовательские исключения, соответствующие лучшим образцам программного кода .NET, на самом деле будут отличаться только именами, поэтому вам будет приятно узнать, что в Visual Studio 2005 предлагается шаблон программного кода под названием "Exception" (рис. 6.5), с помощью которого автоматически генерируется новый класс исключения в соответствии с лучшими рекомендациями .NET (шаблоны программного кода обсуждаются в главе 2).

**Обработка множеств исключений**

В простейшем варианте блок try имеет единственный блок catch. Но на практике часто возникает ситуация, когда операторы в рамках блока try способны создавать *множество*возможных исключений. Например, представьте себе, что метод Accelerate() дополнительно генерирует определенное библиотекой базовых классов исключение ArgumentOutOfRangeException, когда вы передаете методу недопустимый параметр (мы предполагаем, что недопустимым считается любое значение, меньшее нуля).

**// Прежде чем продолжить, проверим допустимость аргумента.**

public void Accelerate (int delta) {

 if (delta ‹ 0) throw new ArgumentOutOfRangeException("Скорость должна быть выше нуля!");

}

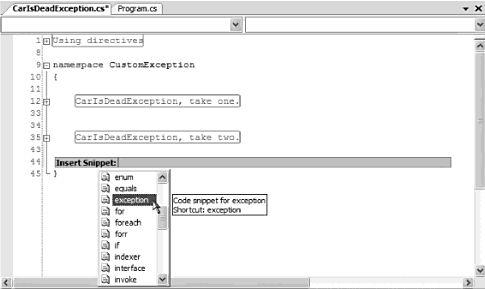


Рис. 6.5. Шаблон программного кода Exception

Логика catch должна соответствовать каждому типу исключений.

static void Main(string [] args) {

 …

**// Здесь учитывается множество исключений.**

try {

 for (int i = 0; i ‹ 10; i++) myCar.Accelerate(10);

} catch(CarIsDeadExeeption e) {

**// Обработка CarIsDeadException.**

} catch (ArgumentOutOfRangeException e) {

**// Обработка ArgumentOutOfRangeException.**

}

При создании множества блоков catch следует учитывать то, что сгенерированное исключение будет обработано "первым подходящим" бликом catch. Чтобы понять, что такое "первый подходящий" блок catch, добавьте в предыдущий фрагмент программного кода еще один блок catch, который будет обрабатывать все исключения после CarIsDeadException и ArgumentOutOfRangeException, выполняя захват System.Exception общего вида, как показано ниже.

**// Этот программный код не компилируется!**

static void Main(string[] args) {

 …

 try {

  for (int i = 0; i ‹ 10; i++) myCar.Accelerate(10);

 } catch(Exception e) {

**// Обработка всех остальных исключений?**

 } catch(CarIsDeadException e) {

**// Обработка CarIsDeadException.**

 } catch(ArgumentOutOfRangeException e) {

**// Обработка ArgumentOutOfRangeException.**

 }

 …

}

Такая логика обработки исключений порождает ошибки компиляции. Проблема в том, что первый блок catch может обработать все*,*что оказывается производным от System.Exception, включая типы CarIsDeadException и ArgumentOutOfRangeException. Таким образом, оставшиеся два блока catch оказываются просто недостижимыми!

Правило, которое следует использовать на практике, заключается в необходимости размещать блоки catch так, чтобы первый блок соответствовал самому "частному" исключению (т.е. самому младшему производному типу в цепочке наследования), а последний блок – самому "общему" (т.е. базовому классу данной цепочки, в данном случае это System.Exception).

Поэтому если вы хотите определить оператор catch, который обработает все ошибки после CarIsDeadException и ArgumentOutOfRangeException, вы должны записать следующее.

**// Этот программный код будет скомпилирован.**

static void Main(string[] args) {

 …

 try {

  for (int i = 0; i ‹ 10; i++) myCar.Accelerate(10);

 } catch(CarIsDeadException e) {

**// Обработка CarIsDeadException.**

 } catch(ArgumentOutOfRangeException) {

**// Обработка ArgumentOutOfRangeException.**

 } catch (Exception e) {

**// Здесь будут обработаны все остальные возможные исключения,**

**// генерируемые операторами в рамках try.**

 }

 …

}

**Общие операторы catch**

В C# также поддерживается "общий" блок catch, который не получает явно объект исключения, генерируемый данным членом.

**// Блок catch общего вида.**

static void Main(string[] args) {

 …

 try {

  for (int i = 0; i ‹ 10; i++) myCar.Accelerate(10);

 } catch {

  Console.WriteLine("Случилось что-то ужасное…");

 }

 …

}

Очевидно, что это не самый информативный способ обработки исключения, поскольку здесь вы не имеете возможности получить содержательную информацию о произошедшей ошибке (например, имя метода, содержимое стека вызовов или пользовательское сообщение). Тем не менее, в C# такая конструкция возможна.

**Генерирование вторичных исключений**

Вы должны знать, что в рамках логики try имеется возможность генерировать исключение для вызывающей стороны, находящейся выше по цепочке вызовов в стеке вызовов. Для этого просто используйте ключевое слово throw в рамках блока сatch. Тем самым исключение будет направлено выше по цепочке логики вызовов, что может быть полезно тогда, когда данный блок catch не имеет возможности полностью обработать возникшую ошибку.

**// Перекладывание ответственности.**

static void Main (string[] args) {

 …

 try {

**// Логика ускорения автомобиля…**}

  catch(CarIsDeadException e) {

**// Частичная обработка ошибки и перенаправление.**

**// Здесь перенаправляется входной объект CarIsDeadException.**

**// Но можно генерировать и другое исключение.**

   throw e;

 }

 …

}

Вы должны понимать, что в данном примере программного кода конечным получателем CarIsDeadException является среда CLR, поскольку здесь вторичное исключение генерируется методом Main(). Поэтому вашему конечному пользователю будет представлено системное диалоговое окно с информацией об ошибке. Как правило, вторично сгенерированное и частично обработанное исключение предъявляется вызывающей стороне, которая имеет возможность обработать его более "грациозно".

**Внутренние исключения**

Вы можете догадываться, что вполне возможно генерировать исключения и во время обработки другого исключения. Например, предположим, что вы обрабатываете CarIsDeadException в рамках конкретного блока catch и в процессе обработки пытаетесь записать след стека в файл carErrors.txt на вашем диске C.

catch(CarlsDeadException e) {

**// Попытка открыть файл carErrors.txt на диске C.**

 FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);

 …

}

Если указанный файл на диске C не найден, попытка вызова File.Open() даст в результате FileNotFoundException. Позже мы рассмотрим пространство имен System.IO и выясним, как перед открытием файла можно программными средствами проверить наличие файла на жестком диске (и предотвратить возможность возникновения исключения). Однако здесь, чтобы сосредоточиться на теме исключений, мы предполагаем, что исключение возникло.

Когда во время обработки одного исключения обнаруживается другое исключение, лучше всего записать новый объект исключения, как "внутреннее исключение" в рамках нового объекта того же типа, что и исходное исключение (язык сломаешь!). Причина, по которой мы должны создавать новый объект исключения, заключается в том, что единственным способом документирования внутреннего исключения оказывается использование параметров конструктора. Рассмотрите следующий фрагмент программного кода.

catch (CarIsDeadException e) {

 try {

  FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);

  …

 } catch(Exception e2*)*{

**// Генерирование исключения, записывающего новое исключение**

**// и сообщение первого исключения.**

  throw new CarIsDeadException(e.Message, e2);

 }

}

Заметьте, что в данном случав мы передали объект FileNotFoundException конструктору CarIsDeadException в виде второго параметра. Сконфигурировав этот новый объект, мы направляем его по стеку вызовов следующему вызывающему объекту, в данном случае по отношению к методу Main().

С учетом того, что после Main() "следующих вызывающих объектов" для обработки исключений нет*,*мы снова должны увидеть диалоговое окно с сообщением об ошибке. Во многом подобно генерированию вторичных исключений, запись внутренних исключений обычно оказывается полезной только тогда, когда вызывающий объект имеет возможность красиво обработать такое исключение. В этом случае логика catch вызывающей стороны может использовать свойство InnerException для получения подробной информации об объекте внутреннего исключения.

**Блок finally**

В рамках try/catch можно также определить необязательный блок finally. Задача блока finally – обеспечить безусловное выполнение некоторого набора операторов программного кода, независимо от наличия или отсутствия исключения (любого типа). Для примера предположим, что вы хотите всегда выключать радио автомобиля перед выходом из Main(), независимо от исключений.

static void Main(string[] args) {

 …

 Car myCar = new Car ('"Zippy", 20);

 myCar.CrankTunes(true);

 try {

**// Логика ускорения автомобиля.**

 } catch (CarIsDeadException e) {

**// Обработка CarIsDeadException.**

 } catch (ArgumentOutOfRangeException e) {

**// Обработка ArgumentOutOfRangeException.**

 } catch(Exception e) {

**// Обработка всех остальных исключений.**

 } finally {

**// Это выполняется всегда. Независимо от исключений.**

  myCar.CrankTunes(false);

 }

}

Если вы не включите в конструкцию блок finally, то радио не будет выключаться, когда обнаруживается исключение (что может быть или не быть проблемой). В более реальном сценарии, когда приходится освобождать ресурсы объектов, закрывать файлы, отключаться от баз данных и т.д., блок finally может гарантировать соответствующую "уборку".

**Что и чем генерируется**

С учетом того, что методы в .NET Framework могут генерировать любое число исключений (в зависимости от обстоятельств), логичным кажется следующий вопрос: "Как узнать, какие именно исключения могут генерироваться тем или иным методом библиотеки базовых классов?" Ответ прост: это можно выяснить в документации .NET Framework 2.0 SDK. В системе справки для каждого метода указаны и исключения, которые может генерировать данный член. В Visual Studio 2005 вам предлагается альтернативный, более быстрый вариант: чтобы увидеть список всех исключений (если таковые имеются), генерируемых данным членом библиотеки базовых классов, достаточно просто задержать указатель мыши на имени члена в окне программного кода (рис. 6.6).

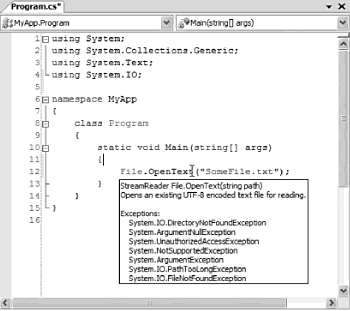


Рис. 6.6. Идентификация исключений, генерируемых данным методом

Тем программистам, которые пришли к изучению .NET, имея опыт работы с Java, важно понять. что, члены типа *не*получают множество исключений из прототипа (другими словами, .NET не поддерживает типизацию исключений). Поэтому вам нет необходимости обрабатывать абсолютно все исключения, генерируемые данным членом. Во многих случаях можно обработать все ошибки путем захвата только System.Exception.

static void Main(string[] args) {

 try {

  File.Open("IDontExist.txt", FileMode.Open);

 } catch(Exception ex) {

  Console.WriteLine(ex.Message);

 }

}

Однако если необходимо обработать конкретные исключения уникальным образом, то придется использовать множество блоков catch, как была показано в данной главе.

**Исключения, оставшиеся без обработки**

Здесь вы можете спросить, что произойдет в том случае, если не обработать исключение, направленное в ваш адрес? Предположим, что программная логика. Main() увеличивает скорость объекта Car выше максимальной скорости в отсутствие программной логики try/catch. Результат игнорирования исключения программой будет очень мешать конечному пользователю, поскольку перед его глазами появится диалоговое окно с информацией о "необработанном исключении". Если на машине установлены инструменты отладки .NET, появится нечто подобное тому, что показано на рис. 6.7 (машина без средств отладки должна отобразить аналогичное, не менее назойливое окно).

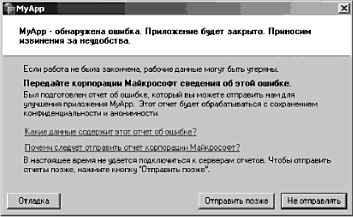


Рис 6.7. Результат игнорирования исключения

**Исходный код.**Проект Custom Exception размещен в подкаталоге, соответствующем главе 6.

**Отладка необработанных исключений в Visual Studio 2005**

В завершение нашего обсуждения следует заметить, что в Visual Studio 2005 предлагается целый ряд инструментов, которые помогают выполнить отладку программ с необработанными пользовательскими исключениями. Снова предположим, что мы увеличили скорость объекта Car выше максимума. Если в Visual Studio запустить сеанс отладки (используя Debug→Start из меню), то выполнение программы автоматически прервется в момент генерирования исключения, оставшегося без обработки. Более того, появится окно (рис. 6.8), в котором будет отображаться значение свойства Message.

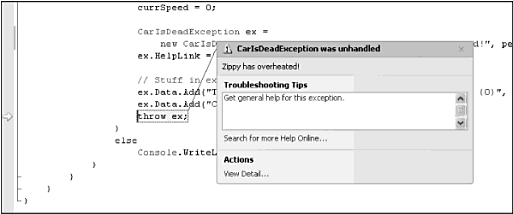


Рис. 6.8. Отладка необработанных пользовательских исключений в Visual Studio 2005

Если щелкнуть на ссылке View Detail (Показать подробности), появится дополнительная информация о состоянии объекта (рис. 6.9).

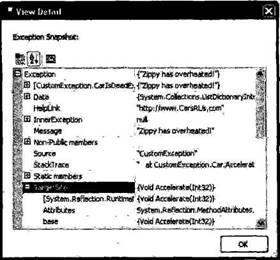


Рис. 6.9. Подробности отладки необработанных пользовательских исключений в Visual Studio 2005

**Замечание.**Если вы не обработаете исключение, сгенерированное методом из библиотеки базовых классов .NET, отладчик Visual Studio 2005 остановит выполнение программы на том операторе, который вызвал метод, создающий проблемы.

**Резюме**

В этой главе мы обсудили роль структурированной обработки исключений. Когда методу требуется отправить объект ошибки вызывающей стороне, этот метод создает, конфигурирует и посылает специальный тип System.Exception, используя для этого ключевое слово C# throw. Вызывающая сторона может обрабатывать поступающие исключения с помощью конструкций, в которых используются ключевое слово catch и необязательный блок finally.

При создании пользовательских исключений вы создаете тип класса, производный от System.ApplicationException, что означает исключение, генерируемое выполняемым приложением. В противоположность этому объекты ошибок, получающиеся из System.SystemException представляют критические (и фатальные) ошибки, генерируемые средой CLR. Наконец, в этой главе были представлены различные инструменты Visual Studio 2005, которые можно использовать при отладке и при создании пользовательских исключений (в соответствии с лучшими образцами .NET).

**ГЛАВА 7. Интерфейсы и коллекции**

В этой главе предлагается рассмотреть тему программирования на основе интерфейсов, чтобы расширять ваши представления об объектно-ориентированном подходе в области разработки приложений. Здесь вы узнаете, как в рамках C# определяются и реализуются интерфейсы, и поймете, в чем заключаются преимущества типов, поддерживающих ''множественное поведение". В процессе обсуждения будет рассмотрен и ряд смежных вопросов – в частности, получение интерфейсных ссылок, явная реализация интерфейсов, а также иерархии интерфейсов.

Часть главы будет посвящена рассмотрению целого ряда интерфейсов, определенных в рамках библиотек базовых классов .NET. Вы увидите, что определенные вами пользовательские типы тоже можно встраивать в эти предопределенные интерфейсы, чтобы обеспечить поддержку специальных функций, таких, как клонирование, перечисление и сортировка объектов.

Чтобы продемонстрировать, как интерфейсы используются в библиотеках базовых классов .NET, в этой главе будут рассмотрено множество встроенных интерфейсов, реализуемых различными классами коллекций (ArrayList, Stack и т.п.), определенными в пространстве имен System.Collections. Информация, представленная здесь, будет необходима для понимания материала главы 10, в которой расcматриваются обобщения .NET и пространство имен Collections.Generiс.

**Определение интерфейсов в C#**

Изложение материала этой главы мы начнем с формального определения типа "интерфейс". Интерфейс – это просто именованная коллекция семантически связанных *абстрактных членов.*Эти члены определяются интерфейсом в зависимости от поведения, которое моделирует данный интерфейс. Интерфейс отражает поведение, которое может поддерживаться данным классом или структурой.

В рамках синтаксиса C# интерфейс определяется с помощью ключевого слова interfасе. В отличие от других типов .NET, интерфейсы никогда не указывают базовый класс (включая System.Object) и для их членов никогда не указываются модификаторы доступа (поскольку все члены интерфейса неявно считаются открытыми). Вот пример пользовательского интерфейса, определенного на языке C#.

**// Этот интерфейс определяет наличие вершин.**

public interface IPointy {

**// Неявно открытый и абстрактный.**

 byte GetNumberOfPoints();

}

**Замечание.**По соглашению имена интерфейсов в библиотеках базовых классов .NET имеют префикс "I" (прописная буква "i" латинского алфавита). При создании пользовательского интерфейса рекомендуется придерживаться аналогичных правил.

Как видите, интерфейс IPointy определяет единственный метод. Однако типы интерфейса в .NET могут также определять любое число свойств. Например, можно определить интерфейс IPointy, в котором используется доступное только для чтения свойство вместо традиционного метода чтения данных.

**// Реализация поведения в виде свойства, доступного только для чтения.**

public interface IPointy {

 byte Points {get;}

}

Вы должны понимать, что типы интерфейса сами по себе совершенно бесполезны, поскольку они представляют собой всего лишь именованные коллекции абстрактных членов. В этой связи вы не можете размещать типы интерфейса так, как это делается с классом или структурой.

**// Создавать типы интерфейса с помощью "new" не допускается.**

static void Main(string[] args) {

 IPointy p = new IPointy(); **// Ошибка компиляции!**

}

Интерфейсы не приносят пользы, если они не реализованы некоторым классом или структурой. Здесь IPointy является интерфейсом, отражающим "наличие вершин". Такое поведение может быть полезным в иерархии форм, построенной нами в главе 4. Идея очень проста: некоторые классы в иерархии форм имеют вершины (например, Hexagon – шестиугольник), а другие (например, Circle – круг) вершин не имеют. Реализовав интерфейс IPointy в Hexagon и Triangle, вы можете предполагать, что оба класса поддерживают общий тип поведения, а поэтому и общее множество членов.

**Реализация интерфейсов в C#**

Чтобы расширить функциональные возможности класса (или структуры) путем поддержки типов интерфейса, нужно просто указать в определении класса (или структуры) список соответствующих типов, разделив их запятыми. Непосредственный базовый класс должен быть *первым элементом*в списке, следующим после операции, обозначаемой двоеточием. Когда тип класса получается непосредственно из System.Object, можно указать только список интерфейсов, поддерживаемым классом, поскольку при отсутствии явного указании компилятор C# получает типы именно из System.Object. Точно так же, поскольку структуры всегда получаются из System.ValueType (см. главу 3), можно указать только интерфейсы в списке, следующем непосредственно после определения структуры. Рассмотрите следующие примеры.

**// Этот класс является производным System.Object.**

**// и реализует один интерфейс.**

public сlаss SomeClass: ISomeInterface {…}

**// Этот класс является производным System.Object**

**// и реализует один интерфейс.**

public class MyClass: object, ISomeInterface {…}

**// Этот класс является производным пользовательского базового класса**

**// и реализует один интерфейс.**

public class AnotherClass: MyBaseClass, ISomeInterface {…}

**// Эта структура является производной System.ValueType**

**// и реализует два интерфейса.**

public struct SomeStruct: ISomeInterface, IPointy

Вы должны понимать, что реализация интерфейса подчиняется принципу "все или ничего". Тип, реализующий интерфейс, не может обеспечивать селективную поддержку членов этого интерфейса. Если интерфейс IPointy определяет единственное свойство, то для выбора вариантов нет. Однако если реализовать интерфейс, определяющий десять членов, то соответствующий тип будет обязан конкретизировать все десять абстрактных элементов.

Так или иначе, вот вам пример реализации обновленной иерархии форм (и обратите внимание на новый тип класса Triangle – треугольник).

**// Hexagon теперь реализует IPointy.**

public class Hexagon: Shape, IPointy {

 public Hexagon() {}

 public Hexagon(string name): base (name) {}

 public override void Draw() { Console.WriteLine("Отображение шестиугольника {0} ", PetName); }

**// Реализация IPointy.**

 public byte Points {

  get { return 6; }

 }

}

**// Новый производный класс Triangle, полученный из Shape.**

public class Triangle: Shape, **IPointy**{

 public Triangle() {}

 public Triangle(string name): base(name) {}

 public override void Draw() { Console.WriteLine("Отображение треугольника {0} ", PetName); }

**// Реализация IPointy.**

 public byte Points {

  get { return 3; }

 }

}

Теперь каждый класс при необходимости возвратит вызывающей стороне число вершин. Чтобы резюмировать сказанное, рассмотрите диаграмму на рис. 7.1, которая была получена в Visual Studio 2005 и иллюстрирует совместимые по интерфейсу IPointy классы, используя популярное обозначение интерфейса знаком "леденца на палочке".

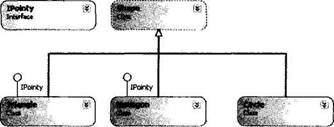


Рис. 7.1. Иерархия форм (теперь с интерфейсами)

**Интерфейсы в сравнении с абстрактными базовыми классами**

С учетом знаний, полученных в главе 4, вы можете спросить, какова причина выдвижения типов интерфейса на первое место. Ведь в C# позволяется строить абстрактные типы класса, содержащие абстрактные методы. И, подобно интерфейсу, при получении класса из абстрактного базового класса, класс тоже обязан определить детали абстрактных методов (если, конечно, производный класс не объявляется абстрактным). Однако возможности абстрактных базовых классов выходят далеко за рамки простого определения группы абстрактных методов. Они могут определять открытые, приватные и защищенные данные состояния, а также любое число конкретных методов, которые оказываются доступными через подклассы.

Интерфейсы, с другой стороны, – это чистый протокол. Интерфейсы *никогда*не определяют данные состояния и *никогда* не обеспечивают реализацию методов (при попытке сделать это вы получите ошибку компиляции).

public interface IAmABadInterface {

**// Ошибка! Интерфейс не может определять данные!**

 int myInt = 0;

 // **Ошибка! Допускается только абстрактные члены!**

 void MyMethod() {Console.WriteLine("Фи!");}

}

Типы интерфейса оказываются полезными и в свете того, что в C# (и других языках .NET) не поддерживается множественное наследование, а основанный на интерфейсах протокол позволяет типу поддерживать множество вариантов поведения, избегая при этом проблем, возникающих при наследовании от множества базовых классов.

Еще более важно то, что программирование на основе интерфейсов обеспечивает альтернативный способ реализаций полиморфного, поведения. Хотя множество классов (или структур) реализует один и тот же интерфейс своими собственными способами, вы имеете возможность обращаться со всеми типами по одной схеме. Чуть позже вы убедитесь, что интерфейсы исключительно полиморфны, потому что с их помощью возможность демонстрировать идентичное поведение получают типы, не связанные классическим наследованием.

**Вызов членов интерфейса на уровне объекта**

Теперь, когда у вас есть набор типов, поддерживающих интерфейс Pointy, следующей задачей оказывается доступ я новым функциональным возможностям. Самым простым способом обеспечения доступа к функциональным возможностям данного интерфейса является непосредственный вызов методов на уровне объектов. Например:

static void Main(string[] args) {

**// вызов члена Points интерфейса IPointy.**

 Hexagon hex = new Hexagon();

 Console.WriteLine("Вершин: {0}", hex.Points);

 Console.ReadLine();

}

Этот подход прекрасно работает в данном конкретном случае, поскольку вы знаете, что тип Hexagon реализует упомянутый интерфейс. Однако в других случаях во время компиляции вы не сможете определить, какие интерфейсы поддерживаются данным типом. Предположим, например, что у нас есть массив из 50 типов, соответствующих Shape, но только некоторые из них поддерживают IPointy. Очевидно, что если вы попытаетесь вызвать свойство Points для типа, в котором IPointy не реализован, вы получите ошибку компиляции. Возникает следующий вопрос: "Как динамически получить информацию о множестве интерфейсов, поддерживаемых данным типом?"

Выяснить во время выполнения, поддерживает ли данный тип конкретный интерфейс можно, например, с помощью явного вызова. Если тип не поддерживает запрошенный интерфейс, вы получите исключение InvalidCastException. Чтобы "изящно" обработать эту возможность, используйте структурированную обработку исключений, например:

static void Main(string[] args) {

 …

**// Возможный захват исключения InvalidCastException.**

 Circle с *=*new Circle ("Lisa");

 IPointу itfPt;

 try {

  itfPt = (IPointy)c;

  Console.WriteLine(itfPt.Points);

 } catch (InvalidCastException e) {

  Console.WriteLine(e.Message);

 }

 Console.ReadLine();

}

Итак, можно использовать логику try/catch и надеяться на удачу, но лучше еще до вызова членов интерфейса определить, какие интерфейсы поддерживаются. Мы рассмотрим два варианта такой тактики.

**Получение интерфейсных ссылок: ключевое слово as**

Второй способ проверить поддержку интерфейса для данного типа предполагает использование ключевого слова as, о котором уже шла речь в главе 4. Если объект можно интерпретировать, как указанный интерфейс, будет возвращена ссылка на интерфейс. Если нет – вы получите null.

static void Main(string[] args) {

 …

**// Можно ли интерпретировать hex2, как IPointy?**

 Hexagon hex2 = new Hexagon("Peter");

 IPointy itfPt2 = hex2 as IPointy;

 if (itfPt2 != null) Console.WriteLine("Вершин: {0}"*,*itfPt2.Points);

 else Console.WriteLine("ОЙ! Вершин не видно…");

}

Обратите внимание на то, что при использовании ключевого слова as не возникает необходимости использовать логику try/catch, поскольку в том случае, когда ссылка оказывается непустой, вы гарантированно будете иметь действительную ссылку на интерфейс.

**Получение интерфейсных ссылок: ключевое слово is**

Можно также проверить реализацию интерфейса с помощью ключевого слова is. Если соответствующий объект не совместим указанным интерфейсом, будет возвращено значение false. А если тип совместим с интерфейсом, вы можете смело вызвать его члены без использования логики try/catch.

Для примера предположим, что мы изменили массив типов Shape так, что теперь некоторые его члены реализуют IPointy. Вот как с помощью ключевого слова is можно выяснить, какие из элементов в массиве поддерживают этот интерфейс.

static void Main(string[] args) {

 …

 Shape[] s = {new Hexagon(), new Circle(), new Triangle("Joe"), new Circle("JoJo")};

 for (int i = 0; i ‹ s.Length; i++) {

**// Напомним, что базовый класс Shape определяет абстрактный**

**// член Draw(), поэтому все формы могут отображать себя.**

  s[i].Draw()

**// Кто с вершинами?**

  if (s[i] is IPointy) Console.WriteLine("-› Вершин: {0} ", ((IPointy)s[i]).Points);

  else Console.WriteLine("-› {0} без вершин!", s[i].PetName);

 }

}

Соответствующий вывод показан на рис. 7.2.

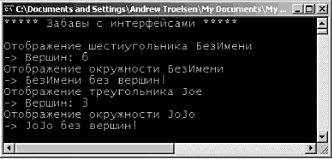


Рис 7.2. Динамическое обнаружение реализованных интерфейсов

**Интерфейсы в качестве параметров**

Поскольку интерфейсы являются полноценными типами .NET, вы можете конструировать методы, которые будут использовать интерфейсы, как параметры. Для примера предположим, что мы определили другой интерфейс с именем IDraw3D.

**// Моделируем возможность отображения типа в пространстве.**

public interface IDraw3D {

 void Draw3D();

}

Предположим также, что две из наших трех форм (Circle и Hexagon) сконфигурированы для поддержки этого нового поведения.

**// Circle поддерживает IDraw3D.**

public class Circle: Shape, **IDraw3D**{

 …

 public void Draw3D() {

  Console.WriteLine("3D-отображение окружности!");

 }

}

**// Hexagon поддерживает IPointy и IDraw3D.**

public class Hexagon: Shape, IPointy, **IDraw3D**{

 …

 public void Draw3D() { Console.WriteLine ("3D-отображение шестиугольника!"); }

}

На рис. 7.3 показана соответствующая обновленная диаграмма классов, полученная в Visual Studio 2005.

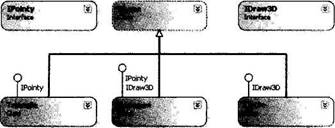


Рис. 7.3. Обновленная иерархия форм

Если определить метод, использующий интерфейс IDraw3D в виде параметра, вы получите возможность передать *любой*объект, реализующий IDraw3D (но если вы попытаетесь передать тип, не поддерживающий нужный интерфейс, будет сгенерирована ошибка компиляции). Рассмотрим следующий фрагмент программного кода.

**// Создание нескольких форм.**

**// Если это возможно, их отображение в трехмерном виде.**

public class Program {

**// Отображение форм, поддерживающих IDraw3D.**

 public static void DrawIn3D(IDraw3D itf3d) {

  Console.WriteLine("-› Отображение IDraw3D-совместимого типа");

  itf3d.Draw3D();

 }

 static void Main() {

  Shape [] s = {new Hexagon(), new Circle(), new Triangle("Joe"), new Circle("JoJo")};

  for (int i = 0; i ‹ s.Length; i++) {

   …

**// Можно ли отобразить в 3D-виде?**

   if (s[i] is IDraw3D) DrawIn3D((IDraw3D)s[i]);

  }

 }

}

Обратите внимание на то, "что треугольник не отображается, поскольку он не является IDraw3D-совместимым (рис. 7.4).

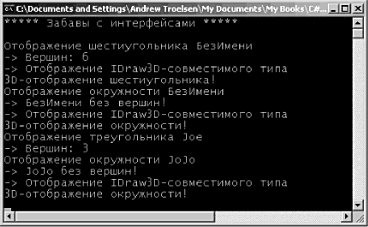


Рис.7.4. Интерфейсы в качестве параметров

**Интерфейсы в качестве возвращаемых значений**

Интерфейсы можно использовать и в качестве возвращаемых значений методов. Например, можно создать метод, который берет любой System.Object, проверяет на совместимость с IPointy и возвращает ссылку на извлеченный интерфейс.

**// Этот метод проверяет соответствие IPointy и, если это возможно,**

**// возвращает ссылку на интерфейс.**

static IPointy ExtractPointyness(object o) {

 if (o is IPointy) return (IPointy*)*o;

 else return null;

}

С этим методом можно взаимодействовать так, как предлагается ниже.

static void Main(string[] args) {

**// Попытка извлечь IPointy из объекта Car.**

 Car myCar = new Car();

 IPointy itfPt = **ExtractPointyness**(myCar);

 if (itfPt!= null) Console.WriteLine("Объект имеет {0} вершин.", itfPt.Points);

 else Console.WriteLine("Этот объект не реализует IPointy");

};

**Массивы интерфейсных типов**

Следует понимать, что один и тот же интерфейс может реализовываться многими типами, даже если эти типы не находятся в рамках одной иерархии классов. В результате можно получать очень мощные программные конструкции, Предположим, например, что мы построили одну иерархию классов для описания кухонной посуды, а другую – для описания садового инвентаря.

Эти иерархии абсолютно не связаны между собой с точки зрения классического наследования, но с ними можно обращаться полиморфно, используя программирование на основе интерфейсов. Для иллюстрации предположим, что у нас есть массив объектов, совместимых с IPointy. При условии, что все объекты этого массива поддерживают один интерфейс, вы можете обходиться с каждым объектом, как с IPointy-совместимым объектом, несмотря на абсолютную несовместимость иерархий классов.

static void Main(string[] args) {

**// Этот массив может содержать только типы,**

**// реализующие интерфейс IPointy.**

 IPointy[] myPointyObjects = {new Hexagon(), new Knife(), new Triangle(), new Fork(), new PitchFork()};

 for (int i = 0; i ‹ myPointyObjects.Length; i++) Console.WriteLine("Объект имеет {0} вершин", myPointyObjects[i].Points);

}

**Замечание.**С учетом общеязыковой природы .NET важно подчеркнуть, что можно определить интерфейс на одном языке (C#), а реализовать его на другом (VB .NET). Но чтобы выяснить, как это сделать, нам потребуется понимание структуры компоновочных блоков .NET, что является темой обсуждения главы 11.

**Явная реализация интерфейса**

В определении IDraw3D мы были вынуждены назвать наш единственный метод Draw3D(), чтобы избежать конфликта с абстрактным методом Draw(), определенным в базовом классе Shape. Такое определение интерфейса вполне допустимо, но более естественным именем для метода было бы Draw().

**// Изменение имени с "Draw3D" на "Draw"**.

public interface IDraw3D {

 void **Draw**();

}

Если вносить такое изменение, то потребуется также обновить нашу реализацию DrawIn3D().

public static void DrawIn3D(IDraw3D itf3d) {

 Console.WriteLine("-› Отображение IDraw3D-совместимоuо типа");

 itf3d.**Draw**();

}

Теперь предположим, что мы определили новый класс Line (линия), который получается из абстрактного класса Shape и реализует iDraw3D (оба из них теперь определяют одинаково названные абстрактные методы Draw()).

**// Проблемы? Это зависит.**…

public class Line: **Shape, IDraw3D**{

 public override void Draw() {

  Console.WriteLine("Отображение линии…");

 }

}

Класс Line компилируется беспрепятственно. Рассмотрим следующую логику Main().

static void Main(string[] args) {

 …

**// Вызов Draw**().

 Line myLine = new Line();

 myLine.Draw();

**// Вызов той же реализации Draw()!**

 IDraw3D itfDraw3d = (IDraw3D)myLine;

 itfDraw3d.Draw();

}

С учетом того, что вы уже знаете о базовом классе Shape и интерфейсе IDraw3D, это выглядит так как будто вы вызываете два варианта метода Draw() (один с объектного уровня, а другой – с помощью интерфейсной ссылки). Однако компилятор способен вызывать одну и ту же реализацию и с помощью интерфейса, и с помощью объектной ссылки, поскольку абстрактный базовый класс Shape и интерфейс IDraw3D имеют одинаково названные члены. Это может оказаться проблемой, когда вы хотите, чтобы метод IDraw3D.Draw() представлял тип во всей трехмерной (3D) "красе", а не в неказистом двухмерном представлении переопределённого метода Shape.Draw().

Теперь рассмотрим родственную проблему. Что делать, если вам нужно гарантировать, что методы, определенные данным интерфейсом, будут доступны только с помощью интерфейсных ссылок, а не объектных? В настоящий момент члены, определенные интерфейсом IPointy, доступны как с помощью объектных ссылок, так и по ссылке на IPointy.

Ответ на оба вопроса дает *явная реализация интерфейса.*Используя этот подход, вы можете гарантировать, что пользователь объекта сможет получить доступ К методам, определенным данным интерфейсом, только с помощью правильной интерфейсной ссылки, не допуская при этом конфликта имен. Для примера рассмотрите следующий обновленный класс Line (предполагая, что вы соответствующим образом обновили Hexagon и Circle).

**// Используя явную реализацию метода, можно указать**

**// другие реализации Draw() .**

public class Line: Shape, IDraw3D {

**// Этот метод можно вызвать только ссылкой на интерфейс IDraw3D.**

 void IDraw3D.Draw() { Console.WriteLine("Отображение ЗD-линии…"); }

**// Это можно вызвать только на уровне объекта.**

 public override void Draw() { Console.WriteLine("Отображение линии…"); }

 …

}

Как видите, при явной реализации члена интерфейса общий шаблон выглядит так: *возвращаемоеЗначение ИмяИнтерфейса.ИмяМетода(аргументы).*Здесь есть несколько "подводных камней", о которых следует знать. Прежде всего, не допускается определять явно реализуемые члены с модификаторами доступа. Например, следующий синтаксис недопустим.

**// Нет! Это недопустимо.**

public class Line: Shape, IDraw3D {

**public** void IDraw3D.Draw() { **// ‹= Ошибка!**

  Console.WriteLine("Отображение 3D-линии…");

 }

 …

}

Главной причиной использования явной реализации метода интерфейса является необходимость "привязки" соответствующего метода интерфейса к уровню интерфейса. Если добавить ключевое слово public, то это будет означать, что данный метод является членом открытого сектора класса, и "привязка" будет отменена. Тогда вызывающая сторона сможет вызывать только метод Draw(), определенный базовым классом Shape на объектном уровне.

**// Здесь вызывается переопределенный метод Shape.Draw().**

Line myLine = new Line();

myLine.Draw();

Чтобы вызвать метод Draw(), определенный с помощью IDraw3D, мы должны явно получить интерфейсную ссылку, используя любой из ранее указанных подходов. Например:

**// Это обеспечит вызов метода IDraw3D.Draw().**

Line myLine = new Line();

IDraw3D i3d = (IDraw3D) myLine;

i3d.Draw();

**Разрешение конфликтов имен**

Явная реализаций интерфейса может оказаться очень полезной тогда, когда реализуются несколько интерфейсов, содержащих идентичные члены, Предположим. например, что вы создали класс, реализующий следующие новые типы интерфейса.

**// Три интерфейса, определяющие методы с одинаковыми именами.**

public interface **IDraw**{

 void Draw();

}

public interface **IDrawToPrinter**{

 void Draw();

}

Если вы захотите построить класс с именем SuperImage (суперизображение), поддерживающий базовую визуализацию (IDraw), 3D-визуализацию (IDraw3D), а также сервис печати (IDrawToPrinter), то единственным способом обеспечить уникальную реализацию для каждого метода будет использование явной реализации интерфейса.

**// Не выводится из Shape, но вводит конфликт имен.**

public class SuperImage: IDraw, IDrawToPrinter, IDraw3D {

 void**IDraw.Draw()**{/\* Логика базовой визуализации. \*/}

 void **IDrawToPrinter.Draw()**{/\* Логика печати. \*/}

 void **IDraw3D.Draw()**{/\* Логика 3D-визуализации. \*/}

}

Исходный код. Проект CustomInterface размешен в подкаталоге, соответствующем главе 7.

**Построение иерархии интерфейсов**

Продолжим наше обсуждение вопросов создания пользовательских интерфейсов и рассмотрим тему *иерархии интерфейсов.*Вы знаете, что класс может выступать в роли базового класса для других классов (которые, в свою очередь, тоже могут быть базовыми классами для других классов), но точно так же можно строить отношения наследования и среди интерфейсов. Как и следует ожидать, интерфейс на вершине иерархии определяет общее поведение, а интерфейсы, находящиеся на более низких уровнях, "уточняют" это поведение. Для примера рассмотрите следующую иерархию интерфейсов.

**// Базовый интерфейс.**

public interface IDrawable { void Draw(); }

public interface IPrintable: **IDrawable**{ void Print(); }

public interface IMetaFileRender: **IPrintable**{ void Render(); }

Соответствующая цепочка наследования показана на рис. 7.5.

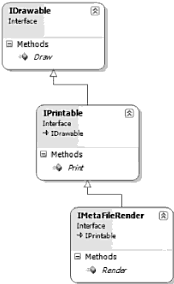


Рис. 7.5. Иерархия интерфейсов

Теперь, если некоторый класс должен поддерживать все варианты поведения, заданные в рамках этой иерархии интерфейсов, то этот класс должен выводиться из интерфейса, лежащего в основе иерархии (в данном случае это IMetaFileRender). Все методы, определенные базовым интерфейсом (или интерфейсами), автоматически переносятся в определение. Например:

**// Этот класс поддерживает IDrawable, IPrintable и IMetaFileRender**.

public class SuperImage: IMetaFileRender {

 public void Draw() { Console.WriteLine("Базовая логика визуализации."); }

 public void Print() { Console.WriteLine("Вывод на принтер."); }

 public void Render() { Console WriteLine("Вывод в метафайл."); }

}

Вот пример вывода каждого интерфейса из экземпляра SuperImage.

**// Использование интерфейсов.**

static void Main(string[] args) {

 SuperImage si = new SuperImage();

**// Получение IDrawable.**

 IDrawable itfDraw = (IDrawable)si;

 itfDraw.Draw();

**// Получение IMetaFileRender, который использует все методы,**

**// определенные выше по цепочке интерфейсов.**

 if (itfDraw is IMetaFileRender) {

  IMetaFileRender itfMF = (IMetaFileRender)itfDraw;

  itfMF.Render();

  itfMF.Print()*;*

 }

 Console.ReadLine();

}

**Интерфейсы с множеством базовых интерфейсов**

При построении иерархии интерфейсов вполне допустимо создавать интерфейсы, которые оказываются производными от нескольких базовых интерфейсов. Однако напомним, что *нельзя*строить классы, которые будут производными от нескольких базовых классов. Для примера предположим, что вы строите набор интерфейсов, моделирующих поведение автомобиля.

public interface ICar { void Drive(); }

public interface IUnderwaterCar { void Dive(); }

// **Здесь интерфейс имеет ДВА базовых интерфейса.**

public interface IJamesBondCar: **ICar, IUnderwaterCar**{ void TurboBoost(); }

На рис. 7.6 показана соответствующая цепочка интерфейсов.

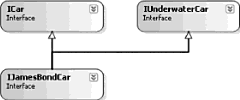


Рис. 7.6. Общая система типов (CTS) допускает множественное наследование интерфейсных типов

При построении класса, реализующего IJamesBondCar (машина Джеймса Бонда), вы должны реализовать TurboBoost(), Dive() и Drive().

public class JamesBondCar: IJamesBondCar {

 public void Drive() { Console.WriteLine("Ускорение…"); }

 public void Dive() { Console.WriteLine("Погружение…"); }

 public void TurboBoost() { Console.WriteLine{"Взлет!"); }

}

Этот специализированный автомобиль можно использовать так, как и ожидается.

static void Main(string[] args) {

 …

 JamesBоndCar j = new JamesBondCar();

 j.Drive();

 j.TurboBoost();

 j.Dive();

}

**Реализация интерфейсов в Visual Studio 2005**

Программирование на основе интерфейсов – это мощная техника программирования, но процесс реализации интерфейсов может предполагать ввод программного кода вручную в очень больших объемах. Поскольку интерфейсы представляют собой именованные наборы абстрактных членов, вам придется вводить программный код для *каждого*метода интерфейса и *каждого*класса, поддерживающего соответствующее поведение.

Вы не ошибаетесь, если предполагаете, что в Visual Studio 2005 имеются различные средства автоматизации для решения задач реализации интерфейсов. Предположим, что нам нужно реализовать интерфейс ICar для нового класса с именем MiniVan. По завершении ввода имени интерфейса (или при помещении указателя мыши на имя интерфейса в окне программного кода) вы обнаружите, что под первой буквой имени появился так называемый "смарт-тег". При щелчке на нем раскрывается список, предлагающий реализовать интерфейс явно или неявно (рис. 7.7).

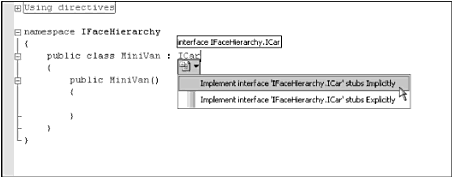


Рис. 7.7. Реализация интерфейсов в Visual Studio 2005

После выбора нужной вам опции Visual Studio 2005 сгенерирует программный код заглушки (в рамках соответствующей именованной области программного кода), который вы затем можете изменить (обратите внимание на то, что по умолчанию реализация предлагает исключение System.Exception).

namespace IFaceHierarchy {

 public class MiniVan: ICar {

  public MiniVan() {}

#region ICar Members

  public void Drive() {

   new Exception("The method or operation is not implemented.");

  }

#endregion

 }

}

Теперь, после обсуждения специфики построения и реализации пользовательских интерфейсов, мы посвятим остаток главы рассмотрению встроенных интерфейсов, содержащихся в библиотеках базовых классов .NET.

**Исходный код**. Проект IFaceHierarchy размещен в подкаталоге, соответствующем главе 7.

**Создание перечислимых типов (Enumerable и IEnumerator)**

Чтобы перейти к иллюстрации процесса реализации существующих интерфейсов .NET, нужно выяснить роль IEnumerable и IEnumerator. Предположим, что у нас есть класс Garage (гараж), содержащий некоторый набор типов Car (см. главу б), хранимых в виде System.Array.

**// Garage содержит набор объектов Car.**

public class Garage {

 private Car[] carArray;

 // **Начальное наполнение объектами Car.**

 public Garage() {

  carArray = new Car[4];

  carArray[0] = new Car("Rusty", 30);

  carArray[1] = new Car("Clunker", 55);

  carArray[2] = new Car("Zippy", 30);

  carArray[3] = new Car("Fred", 30);

 }

}

Было бы удобно выполнить проход по элементам, содержащимся в объекте Garage, используя конструкцию C# foreach.

**// Это кажется разумным…**

public class Program {

static void Main(string[] args) {

 Garage carLot = new Garage();

**// Для каждого объекта Car в коллекции?**

 foreach (Car c in carLot) {

  Console.WriteLine("{0} имеет скорость {1} км/ч", с.PetName, с.CurrSpeed);)

 }

}

Но, как это ни печально, компилятор сообщит вам, что класс Garage не реализует метод GetEnumerator(). Этот метод формально определен интерфейсом IEnumerable, находящимся в "недрах" пространства имен System.Collections. Объекты, поддерживающие соответствующий вариант поведения, декларируют, что они могут раскрыть содержащиеся в них элементы вызывающей стороне.

**// Этот интерфейс информирует вызывающую сторону о том,**

**// что элементы объекта перечислимы.**

public interface IEnumerable {

 IEnumerator GetEnumerator();

}

Как видите, метод GetEnumerator() должен возвращать ссылку на другой интерфейс – интерфейс c именем System.Collections.IEnumerator. Этот интерфейс предлагает инфраструктуру, которая позволяет вызывающей стороне выполнить цикл по объектам, содержащимся в IEnumerable-совместимом контейнере.

**// Этот интерфейс позволяет вызывающей стороне**

**// получить внутренние элементы контейнера.**

public interface IEnumerator {

 bool MoveNext(); **// Сдвинуть на позицию вперед.**

 object Current { get;} **// Прочитать (свойство только для чтения).**

 void Reset(); **// Сдвинуть в начальную позицию.**

}

Чтобы обеспечить поддержку указанных интерфейсов типом Garage, можно пойти по длинному пути реализации каждого метода вручную. Конечно, ничто не запрещает указать свои версии GetEnumerator(), MoveNext()*,*Current и Reset(), но есть и более простой путь. Поскольку тип System.Array, как и многие другие типы, уже реализован в IEnumerable и IEnumerator, вы можете просто делегировать запрос к System.Array, как показано ниже.

using System.Collections;

public class Garage: **IEnumerable**{

**// В System.Array уже есть реализация IEnumerator!**

 private Car[] carArray;

 public Cars() {

  carArray = new Car[4];

  carArray[0] = new Car("FeeFee", 200, 0);

  carArray[l] = new Car("Clunker", 90, 0);

  carArray[2] = new Car("Zippy, 30, 0);

  carArray[3] = new Car("Fjred", 30, 0);}

  public IEnumerator GetEnumerator() {

**// Возвращает IEnumerator объекта массива.**

  return carArray.GetEnumerator();

 }

}

Теперь, после модификации типа Garage, вы можете использовать этот тип в конструкции foreach без опасений. К тому же, поскольку метод GetEnumerator() определен, как открытый, пользователь объекта тоже может взаимодействовать с типом IEnumerator.

**// Manually work with IEnumerator.**

IEnumerator I = carLot.GetEnumerator();

i.MoveNext();

Car myCar = (Car)i.Current;

Console.WriteLine("{0} имеет скорость {1} км/ч", myCar.PetName, myCar.CurrSpeed);

Если вы предпочтете скрыть функциональные возможности IEnumerable на объектном уровне, то следует использовать явную реализацию интерфейса.

public IEnumerator IEnumerable.GetEnumerator() {

**// Возвращает IEnumerator объекта массива.**

 return carArray.GetEnumerator();

}

**Исходный код.**Проект CustomEnumerator размещен в подкаталоге, соответствующем главе 7.

**Методы итератора в C#**

В .NET 1.x для того, чтобы пользовательские коллекции (такие, как Garage) допускали применение конструкции foreach в операциях, подобных перечислению, реализация интерфейса IEnumerable (и, как правило, интерфейса IEnumerator) была обязательной. В C# 2005 предлагается альтернативный вариант построения типов, позволяющих применение цикла foreach, – с помощью *итераторов.*

В упрощённой интерпретации итератор является членом, указывающим порядок возвращения внутренних элементов контейнера при их обработке с помощью foreach. И хотя метод итератора все равно должен называться GetEnumerator(), а возвращаемое значение все равно должно иметь тип IEnumerator, при таком подходе ваш пользовательский класс уже *не*обязан реализовывать все ожидаемые интерфейсы.

public class Garage { **// Без реализации IEnumerator!**

 private Car[] carArray;

 …

**// Метод итератора.**

 public IEnumerator GetEnumerator() {

  foreach (Car с in carArray) {

   yield return c;

  }

 }

}

Обратите внимание на то, что данная реализация GetEnumerator() осуществляет "проход" по вложенным элементам, используя внутреннюю логику foreach, и возвращает объекты Car вызывающей стороне, используя новую синтаксическую конструкцию yield return. Ключевое слово yield используется для того, чтобы указать значение (или значения), возвращаемые конструкции foreach вызывающей стороны. Когда в программе встречается оператор yield return, сохраняется текущая позиция, и именно с этой позиции выполнение будет продолжено при следующем вызове итератора.

Когда компилятор C# обнаруживает метод итератора, в рамках области видимости соответствующего типа (в данном случае это Garage) динамически генерируется вложенный класс. Этот автоматически сгенерированный класс реализует интерфейсы IEnumerable и IEnumerator и указывает необходимые параметры членов GetEnumerator(), MoveNext(), Reset() и Current. Если теперь загрузить данное приложение в ildasm.exe, то будет видно, что внутренняя реализация GetEnumerator() в объекте Garage использует сгенерированный компилятором тип (который в данном примере получает имя*‹*GetEnumerator*›*d\_\_0).

.method public hidebysig instance class [mscorlib] System.Collections.IEnumerator GetEnumerator() cil managed {

 …

 newobj instance void CustomEnumeratorWithYield.Garage/ '‹**GetEnumerator›d\_\_0**'::.ctor(int32)

 …

} // end of method Garage::GetEnumerator

Явно, что от предложенного здесь определения метода итератора мы не получим большой пользы, поскольку наш тип Garage изначально реализовывал GetEnumerator(), ссылаясь на внутренний тип System.Array. Но синтаксис итератора C# может сэкономить немало времени при построении более "экзотических" пользовательских контейнеров (например, бинарных деревьев), где приходится вручную реализовать интерфейсы IEnumerator и IEnumerable. В любом случае программный код вызывающей стороны при взаимодействии с методом итератора с использованием foreach оказывается одинаковым.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с методами итератора \*\*\*\*\*\n");

 Garage carLot = new Garage();

 foreach (Car с in carLot) {

  Console.WriteLine("{0} имеет скорость {1} км/ч", с.PetName, с.CurrrSpeed);

 }

 Console.ReadLine();

}

**Исходный код**. Проект CustomEnumeratorWifhYield размещен в подкаталоге, соответствующем главе 7.

**Создание клонируемых объектов (ICloneable)**

Вы, должно быть, помните из главы 3, что System.Object определяет член с именем MemberwiseClone(). Указанный метод используется для получения *поверхностной*копии объекта. Пользователи объекта не могут вызвать этот метод непосредственно (поскольку он является защищенным), но сам объект может вызвать этот метод в процессе клонирования. Для примера предположим, что у нас есть класс с именем Point (точка).

**// Класс Point.**

public class Point {

**// Открыты для простоты.**

 public int x, у;

 public Point(int x, int y) { this.x = x; this.у = у; }

 public Point(){}

**// Переопределение Object.ToString().**

 public override string ToString() { return string.Format("X = {0}; Y = {1}", x, у); }

}

С учетом того, что вы уже знаете о ссылочных типах и типах, характеризуемых значениями (см. главу 3), вы должны понимать, что в результате присваивания одной ссылочной переменной другой получаются две ссылки, указывающие на один и тот же объект в памяти. Поэтому следующее присваивание дает две ссылки на один и тот же объект Point в динамической памяти, и модификации любой из этих ссылок будут влиять на этот объект.

static void Main(string[] args) {

**// Две ссылки на один и тот же объект!**

 Point p1 = new Point(50, 50);

 Point p2 = p1;

 р2.х = 0;

 Console.WriteLine(p1);

 Console.WriteLine(p2);

}

Чтобы обеспечить пользовательскому типу возможность возвращать копию этого типа вызывающей стороне, можно реализовать стандартный интерфейс ICloneable. Этот интерфейс определяет единственный метод с именем Clone().

public interface **ICloneable**{

 object Clone();

}

Очевидно, что реализация метода Clone() будет зависеть от объекта. Но базовые функциональные возможности оказываются одинаковыми: это копирование значений членов-переменных в новый экземпляр объекта и возвращение этого экземпляра пользователю. В качестве иллюстрации рассмотрите следующую модификацию класса Point.

**// Теперь Point поддерживает клонирование.**

public class Point: **ICloneable**{

 public int x, y;

 public Point(){}

 public Point (int x, int y) { this.x = x; this.у = у; }

**// Возвращение копии данного объекта.**

 public object Clone() { return new Point(this.x, this.y); }

 public override string ToString() { return String.Format("X = {0}; Y = {1}", x, у); }

}

С помощью указанного подхода можно создавать точные и независимые копии типа Point, как показано в следующем фрагменте программного кода.

static void Main (string[] args) {

**// Обратите внимание, Clone() возвращает объект общего типа.**

**// Для получении производного типа используйте явное преобразование.**

 Point р3 = new Point(100, 100);

 Point р4 = (Point)р3.Clone();

**// Изменение p4.х (это не изменит р3.х).**

 р4.х = 0;

**// Вывод объектов.**

 Console.WriteLine(р3);

 Console.WriteLine(p4);

}

Текущая реализация Point решает все поставленные задачи, но вы можете немного усовершенствовать процесс. Введу того, что тип Point не содержит переменных ссылочного типа, можно упростить реализацию метода Clone(), как показано ниже.

public object Clone() {

**// Скопировать все поля Point "почленно"**.

 return this.MemberwiseClone();

}

При наличии в Point членов-переменных ссылочного типа метод MemberwiseClone() скопирует ссылки на соответствующие объекты (т.е. выполнит *поверхностное копирование).*Чтобы обеспечить поддержку полного копирования объектов, вы должны в процессе клонирования создать новый экземпляр для каждой переменной ссылочного типа. Соответствующий пример мы сейчас рассмотрим.

**Пример клонирования**

Предположим, что класс Point содержит член ссылочного типа с именем PointDescription, обеспечивающий поддержку "понятного" имени объекта Point и его идентификационного номера в виде System.Guid (еcли у вас нет опыта применения COM, знайте, что GUID – глобально уникальный идентификатор – это статистически уникальное 128-разрядное значение). Вот соответствующая реализация.

**// Этот класс описывает точку.**

public class PointDescription {

**// Открыты для простоты.**

 public string petName;

 public Guid pointID;

 public PointDescription() {

  this.petName = "Без имени";

  pointID = Guid.NewGuid();

 }

}

При этом для учета новых элементов состояния в самом классе Point следует изменить метод ToString(), а также операторы определения и создания ссылочного типа PointDescription. Чтобы позволить "внешнему миру" указать имя для Point, можно также модифицировать аргументы, передаваемые перегруженному конструктору.

public class Point: ICloneable {

 public int x, y;

 public PointDescription desc = new PointDescription();

 public Point(){}

 public Point (int x, int y) {

  this.x = x;

  this.у = у;

 }

 public Point(int x, int y, string petname) {

  this.x = x;

  this.у = у;

  desc.petName = petname;

 }

 public object Clone() { return this.MemberwiseClone(); }

 public override string ToString() {

  return string.Format("X = {0}; Y = {1}; Имя = (2};\nID = {3}\n", x, y, desc.petName*,*desc.pointID);

 }

}

He забудьте о том. что вы еще не обновили метод Clone(). Поэтому при запросе клонирования объекта пользователем с помощью данной реализации все равно будет получена поверхностная ("почленная") копия. Для примера предположим, что мы обновили метод Main() так, как показано ниже.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с ICloneable \*\*\*\*\*\n");

 Console.WriteLine("Клонирован р3, новый Point сохранен в р4");

 Point p3 = new Point(100, 100, "Jane");

 Point p4 = (Point)p3.Clone();

 Console.WriteLine("До модификации:");

 Console.WriteLine("р3: {0}", р3);

 Console.WriteLine("p4: {0}", p4);

 p4.desc.petName = "Мистер X";

 p4.x = 9;

 Console.WriteLine("Изменены p4.desc.petName и р4.х");

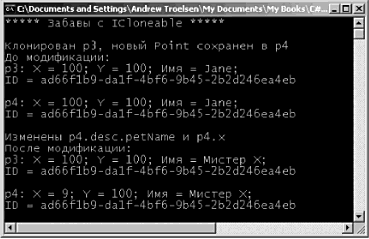
 Console.WriteLine("После модификации: ");

 Console.WriteLine("p3: {0}", р3);

 Console.WriteLine("p4: {0}", p4)*;*

}

На рис. 7.8 показан соответствующий вывод.



Рис*.*7.8. Метод MemberwiseClone() возвращает поверхностную копию объекта

Для того чтобы метод Clone() возвращал полные копии внутренних ссылочных типов, нужно "научить" возвращаемый методом MemberwiseClone() объект учитывать текущее имя объекта Point (тип System.Guid является структурой, так что на самом деле копируются числовые данные). Вот одна из возможных реализаций.

**// Мы должны учесть наличие члена PointDescription.**

public object Clone() {

 Point newPoint = (Point)this.MemberwiseClone();

 PointDescription currentDesc = new PointDescription();

 сurrentDesc.petName = this.desc.petName;

 newPoint.desc = currentDesc;

 return newPoint;

}

Если выполнить приложение теперь, то вы увидите (рис. 7.9), что возвращенный методом Clone() объект Point действительно копирует внутренние ссылочные члены-переменные типа (обратите внимание на то, что здесь p3 и p4 имеют свои уникальные имена).

Итак, в том случае, когда класс или структура содержит только типы, характеризуемые значениями, лучше реализовать метод Clone(), использующий MemberwiseClone(). Однако в том случае, когда пользовательский тип содержит ссылочные типы, вы должны создать новый тип, принимающий во внимание все члены-переменные ссылочного типа.

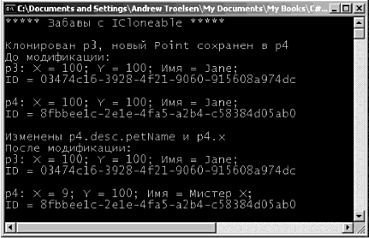


Рис. 7.9. Здесь получена полная копия объекта

**Исходный код.**Проект CloneablePoint размещен в подкаталоге, соответствующем главе 7.

**Создание сравнимых объектов (IComparable)**

Интерфейс System.IComparable определяет поведение, позволяющее сортировать объекты по заданному ключу. Вот формальное определение.

**// Этот интерфейс позволяет объекту указать его связь**

**// с другими подобными объектами.**

public interface IComparable {

 int CompareTo(object o);

}

Предположим теперь, что класс Car поддерживает некоторый внутренний идентификатор (представленный целым числом, хранимым в переменной carID), значение которого можно устанавливать с помощью параметра конструктора и изменять с помощью нового свойства ID. Ниже показана соответствующая модификация типа Car.

public class Car {

 …

 private int carID;

 public int ID {

  get { return carID; }

  set { carID = value; }

 }

 public Car(string name, int currSp, int id) {

  currSpeed = currSp;

  petName = name;

  carID = id;

 }

 …

}

Пользователи объекта могут создать массив типов Car так.

static void Main(string[] args) {

**// Создание массива типов Car.**

 Car[] myAutos = new Car[5];

 myAutos[0] = new Car("Rusty", 80, 1);

 myAutos[1] = new Car("Mary"*,*40, 234);

 myAutos[2] = new Car("Viper", 40, 34);

 myAutos[3] = new Car("Mel", 40, 4);

 myAutos[4] = new Car("Chucky", 40, 5);

}

Вспомним, что класс System.Array определяет статический метод Sort(). Вызвав этот метод для массива встроенных типов (int, short, string и т.д.), можно отсортировать элементы в массиве в числовом или алфавитном порядке, поскольку встроенные типы данных реализуют IComparable. Но что произойдет в том случае, когда методу Sort() будет передан массив типов Car, как показано ниже?

**// Будут ли отсортированы мои автомобили?**

Array.Sort(myAutos);

Запустив этот пример, вы обнаружите, что среда выполнения сгенерирует исключение ArgumentException c сообщением следующего содержания: "Как минимум один объект должен реализовать IComparable". Чтобы позволить сортировку массивов ваших пользовательских типов, вы должны реализовать IComparable. При создании CompareTo() вы должны решить, что должно лежать в основе соответствующей операции упорядочения. Для типа Car самым подходящим "кандидатом" является carID.

**// Последовательность Car можно упорядочить на основе CarID.**

public class Car: **IComparable**{

 …

**// Реализация IComparable.**

 int IComparable.CompareTo(object obj) {

  Car temp = (Car)obj;

  if (this.carID › temp.carID) return 1;

  if(this.carID *‹*temp.carID) return -1;

  else return 0;

 }

}

Как видите, в CompareTo() выполняется сравнение поступившего типа с текущим экземпляром на основе сравнения значений заданных элементов данных. Возвращаемое значение CompareTo() указывает, будет ли данный тип меньше, больше или равен объекту сравнения (см. табл. 7.1).

Таблица 7.1. Возвращаемые значения CompareTo()

|  |  |
| --- | --- |
| **Возвращаемое значение** | **Описание** |
| Любое число, меньшее нуля | В данном порядке сортировки текущий экземпляр размещается до указанного объекта |
| Нуль | Этот экземпляр равен указанному объекту |
| Любое число, большее нуля | В данном порядке сортировки текущий экземпляр размещается после указанного объекта |

Теперь, когда тип Car "умеет" сравнивать себя с подобными объектами, вы можете записать пользовательский программный код следующего вида.

**// Проверка интерфейса IComparable.**

static void Main(string[] args) {

**// Создание массива типов Car.**

**// Вывод исходного массива**.

 Console.WriteLine("Несортированный набор машин:");

 foreach(Car с in myAutos) Console.WriteLine("{0) (1}", с.ID, с.petName);

**// Теперь сортируем их с помощью IComparable.**

 Array.Sort(myAutos);

**// Вывод отсортированного массива.**

 Console.WriteLine("\nУпорядоченный набор машин:");

 foreach(Car с in myAutos) Console.WriteLine("{0} {1}", с.ID, с.petName);

 Console.ReadLine();

}

На рис. 7.10 показан соответствующий вывод.

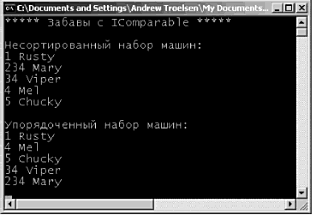


Рис. 7.10. Сравнение автомобилей на основе значений ID

**Сортировка по набору критериев (IComparer)**

В этой версии типа Car в качестве критерия упорядочения мы использовали ID автомобиля, В другом случае для сортировки можно использовать, например, petName (чтобы разместить автомобили в алфавитном порядке их названий). Но что делать, если нужно отсортировать автомобили и по значению ID, и по значению petName? В этом случае вы должны использовать другой стандартный интерфейс, определенный в рамках пространства имен System.Collections, – интерфейс IComparer.

**// Типичный способ сравнения двух объектов.**

interface IComparer {

 int Compare(object o1, object o2);

}

В отличие от IComparable, интерфейс IComparer обычно реализуют не с помощью типов, которые предполагается сортировать (в данном случае это типы Car), а с помощью некоторого набора вспомогательных классов, по одному для каждого порядка сортировки (petName, ID и т.д.). Тип Car (автомобиль) уже "знает", как сравнивать себя с другими автомобилями на основе внутреннего идентификатора ID. Чтобы позволить пользователю объекта отсортировать массив типов Car по значению petName, нам потребуется вспомогательный класс, реализующий IComparer. Вот подходящий для этого программный код.

**// Этот вспомогательный класс используется для сортировки**

**// массива объектов Car по названию.**

using System.Collections;

public class PetNameComparer : **IComparer**{

 public PetNameComparer() {}

**// Проверка названий объектов**.

 int IComраrer.Compare(object o1, object o2) {

  Car t1 = (Car)о1;

  Car t2 = (Car)o2;

  return **String.Compare**(t1.petName, t2.petName);

 }

}

Этот вспомогательный класс можно использовать в программном коде пользователя объекта. Класс System.Array предлагает перегруженный метод Sort(), один из вариантов которого допускает использование объекта, реализующего интерфейс IComparer (рис. 7.11).

static void Main (string[] args) {

 …

**// Теперь сортируем по имени.**

 Array.Sort(myAutos, new РеtNameComparer());

**// Вывод отсортированного массива.**

 Consolе.WriteLine("\nУпорядочение по названию");

 foreach(Car e in myAutos) Console.WriteLine("{0} {1}", c.ID, c.petName);

 …

}

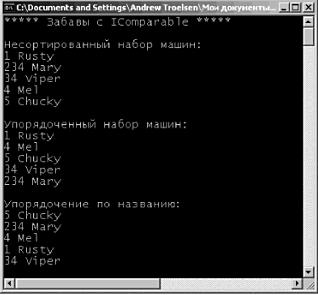


Рис. 7.11. Сортировка автомобилей по названию

**Типы, определяющие сортировку, и пользовательские свойства**

Следует отметить, что с помощью пользовательских статических свойств вы можете помочь пользователю объекта отсортировать типы Car по заданному элементу данных. Предположим, что в класс Car добавлено статическое свойство SortByPetName(), доступное только для чтения и возвращающее экземпляр объекта, реализующего интерфейс IComparer (в данном случае это PetNameComparer).

**// Здесь обеспечивается поддержка пользовательского свойства для**

**// возвращения "правильного" интерфейса IComparer.**

public class Car: **IComparable**{

 …

**// Свойство, возвращающее компаратор SortByPetName**.

 public static IComparer SortByPetName { get { return (IComparer)new PetNameComparer(); } }

}

В программном коде пользователя объекта теперь можно выполнить сортировку по названию, используя ассоциированное свойство без какого бы то ни было "упоминания" специального типа класса PetNameComparer:

**// Сортировка по имени становится немного проще.**

Array.Sort(myAutos, **Car.SortByPetName**);

**Исходный код.**Проект ComparableCar размещен в подкаталоге, соответствующем главе 7.

Теперь вы должны понимать не только то, как определяются и реализуются типы интерфейса, но и то, в чем их польза. Будьте уверены, интерфейсы можно обнаружить в любом из главных пространств имен .NET и в завершение этой главы мы рассмотрим примеры интерфейсов (и базовых классов) из пространства имен System.Collections.

**Интерфейсы из пространства имен System.Collections**

В качестве самого примитивного контейнера может выступать тип System.Array. В главе 3 было показано, что класс System.Array предлагает целый ряд соответствующих возможностей (таких, как инвертирование, сортировка, очистка и перечисление). Но класс Array имеет свои ограничения, и наиболее важным из них является невозможность динамического переопределения размеров при добавлении и удалении элементов. Если для хранения типов необходим более "гибкий" контейнер, лучше использовать типы, определенные в пространстве имен System.Collections (или, в соответствии с рекомендациями главы 10, из пространства имен System.Collections.Generic).

Пространство имен System.Collections определяет целый ряд интерфейсов (и некоторые из них уже использовались в примерах этой главы). Как правило, коллекции классов реализует эти интерфейсы для того, чтобы обеспечить доступ к своему содержимому. В табл. 7.2. приводятся описания основных интерфейсов, относящихся к коллекциям.

Таблица 7.2. Интерфейсы System.Collections

|  |  |
| --- | --- |
| **Интерфейс** | **Описание** |
| ICollection | Определяет общие характеристики (такие, как защищенность счетчиков и цепочек) для типа коллекции |
| IComparer | Позволяет сравнение двух объектов |
| IDictionary | Позволяет объекту представить его содержимое с помощью пар имен и значений |
| IDictionaryEditor | Перечисляет содержимое типа, поддерживающего IDictionary |
| IEnumerable | Возвращает интерфейс IEnumerator для данного объекта |
| IEnumerator | Обеспечивает общую поддержку перечисления подтипов с помощью foreach |
| IHashCodeProvider | Возвращает хеш-код для реализующего типа, используя настраиваемый механизм хеширования |
| IKeyComparer | Этот интерфейс является новым в .NET 2.0). Объединяет функциональные возможности IComparer и IHashCodeProvider, чтобы обеспечить сравнение объектов по их хеш-кодам (если объекты (равны, то они должны возвращать одинаковые хеш-коды) |
| IList | Обеспечивает возможность добавления, удаления и индексирования элементов в списке объектов. Кроме того, с помощью членов этого интерфейса можно выяснить, является ли данный тип-контейнер коллекции доступным только для чтения, и имеет ли он фиксированный размер |

Многие из этих интерфейсов связаны иерархией интерфейсов, в то время как другие являются автономными единицами. На рис. 7.12 показана схема взаимосвязей между указанными типами (напомним, что один интерфейс может быть производным от нескольких интерфейсов).

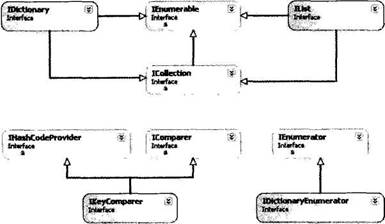


Рис. 7.12. Иерархия интерфейсов System.Collections

**Интерфейс ICollection**

Интерфейс ICollection является простейшим интерфейсом пространства имен System.Collections в том смысле, что этот интерфейс определяет поведение, поддерживаемое любым типом коллекции. По сути, этот интерфейс обеспечивает узкий набор свойств, которые позволяют определить: а) число элементов в контейнере; б) защищенность цепочки контейнера; в) возможность копирования содержимого в тип System.Array. Формально ICollection определяется так, как показано ниже (обратите внимание на то, что ICollection расширяет IEnumerable).

public interface **ICollection**: IEnumerable {

**// Член IEnumerable.**…

 int Count { get; }

 bool IsSynchronized { get; }

 object SyncRoot { get; }

 void CopyTo(Array array, int index);

}

**Интерфейс IDictionary**

Вы, возможно, знаете, что *словарь*- это коллекция, обеспечивающая поддержку пар имен и их значений. Например, можно построить пользовательский тип, реализующий IDictionary, в котором вы сможете сохранить типы Car (значения) с возможностью их последующего восстановления по ID или petName (это примеры имен). Интерфейс IDictionary определяет свойства Keys и Values, а также методы Add(), Remove() и Contains(). Отдельные элементы можно получить c помощью индексатора типа. Вот формальное определение.

public interface **IDictionary**: ICollection, IEnumerable {

 bool IsFixedSize { get; }

 bool IsReadOnly { get; }

 object this [object key] { get; set; }

 ICollection Keys { get; }

 ICollection Values { get; }

 void Add(object key, object value);

 void Clear();

 bool Contains(Object key);

 IDictionaryEnumerator GetEnumerator()*;*

 void Remove(object key);

}

**Интерфейс IDictionaryEnumerator**

При внимательном чтении вы могли заметить, что IDictionary.GetEnumerator() возвращает экземпляр IDictionaryEnumerator. Тип IDictionaryEnumerator – это строго типизованный нумератор, расширяющий IEnumerator путем добавления следующей функциональной возможности.

public interface **IDictionaryEnumerator**: IEnumerator {

**// Методы IEnumerator…**

 DictionaryEntry Entry { get; }

 object Key { get; }

 object Value { get; }

}

Обратите внимание на то, что IDictionaryEnumerator обеспечивает возможность перечисления элементов словаря с помощью общего свойства Entry, которое возвращает тип класса System.Collections.DictionaryEntry. Кроме того, вы можете выполнить цикл по парам имен и значений, используя свойства Key/Value.

**Интерфейс IList**

Последним из ключевых интерфейсов System.Collections является интерфейс IList, который обеспечивает возможность вставки, удаления и индексирования элементов контейнера.

public interface **IList**: ICollection, IEnumerable {

 bool IsFixedSize { get; }

 bool IsReadOnly { get; }

 object this[int index] { get; set; }

 int Add(object value);

 void Clear();

 bool Contains(object value);

 int IndexOf(object value);

 void Insert(int index, object value);

 void Remove(object value);

 void RemoveAt(int index);

}

**Классы из пространства имен System.Collections**

Еще раз подчеркнем, что интерфейсы остаются бесполезными до тех пор, пока они не реализованы соответствующим классом или соответствующей структурой. В табл. 7.3 предлагаются описания основных классов из пространства имен System.Collections вместе с ключевыми интерфейсами, которые этими классами поддерживаются.

Таблица 7.3. Классы System.Collections

|  |  |  |
| --- | --- | --- |
| **Класс** | **Описание** | **Реализуемые интерфейсы** |
| ArrayList | Представляет динамически изменяемый по размерам массив объектов | IList, ICollection, IEnumerable, ICloneable |
| Hashtable | Представляет коллекцию объектов, идентифицируемых по числовому ключу. Пользовательские типы, хранимые в Hashtable, должны обязательно переопределять System.Object.GetHashCode() | IDictionary, ICollection, IEnumerable, ICloneable |
| Queue | Представляет стандартную очередь FIFO (first-in, first-out – первым прибыл, первым обслужен) | ICollection, ICloneable, IEnumerable |
| SortedList | Подобен словарю, но здесь элементы могут быть также доступны по позиции (например, по индексу) | IDictionary, ICollection, IEnumerable, ICloneable |
| Stack | Очередь LIFO (last-in, first-out – последним прибыл, первым обслужен), обеспечивающая функциональные возможности стека | ICollection, ICloneable, IEnumerable |

Вдобавок к этим ключевым типам в System.Collections определяются некоторые менее значительные (в смысле частоты использования) "игроки", такие как BitArray, CaseInsensitiveComparer и CaseInsensitiveHashCodeProvider. Кроме того, это пространство имен определяет небольшой набор абстрактных базовых классов (CollectionBase, ReadOnlyCollectionBase и DictionaryBase), которые могут использоваться для построения строго типизованных контейнеров.

Экспериментируя с типами System.Collections, вы обнаружите, что все они "стремятся" использовать общие функциональные возможности (в этом и заключается суть программирования на основе интерфейсов). Поэтому вместо описания всех членов каждого класса коллекции задачей нашего обсуждения будет демонстрация возможностей взаимодействия с тремя главными типами коллекций - ArrayList. Queue and Stack. Освоив функциональные возможности этих типов, вы без особого труда сможете прийти к пониманию и остальных классов коллекций (особенно если учесть что в файлах справки предлагается исчерпывающая документация для каждого из типов).

**Работа с типом ArrayList**

Тип ArrayList непременно станет для вас наиболее часто используемым типом пространства имей System.Collections, поскольку он позволяет динамически переопределять размеры содержимого. Для иллюстрации базовых возможностей этого типа предлагаем вам рассмотреть следующий программный код, в котором ArrayList используется для манипуляций с набором объектов Car.

static void Main(string[] args) {

**// Создание ArrayList и заполнение исходными значениями.**

 ArrayList carArList = new ArrayList();

 carArList.AddRange(new Car[] {new Car("Fred", 90*,*10), new Car("Mary", 100, 50), new Car("MB", 190, 11)});

 Console.WriteLine("\nЭлементов в carArList: {0}", carArList.Count);

**// Печать текущих значений.**

 foreach(Car с in carArList) Console.WriteLine("Имя автомобиля: {0}", c.petName);

**// Вставка нового элемента.**

 Console.WriteLine("\n-›Добавление нового Car.");

 carArList.Insert(2, new Car("TheNewCar", 0, 12));

 Console.WriteLine("Элементов в carArList: {0}", carArList.Count);

**// Получение массива объектов из ArrayList и снова печать.**

 object[] arrayOfCars = carArList.ToArray();

 for (int i = 0; i ‹ arrayOfCar.Length; i++) {

  Console.WriteLine("Имя автомобиля: {0}", ((Car) arrayOfCars[i]).petName);

 }

}

Здесь для добавления в коллекцию ArrayList набора типов Car используется метод AddRange() (который, по сути, заменяет n-кратный вызов метода Add()). После вывода информации о числе элементов в коллекции (и после цикла по всем элементам для получения имен) вызывается метод Insert(). Как видите, Insert() позволяет осуществить вставку нового элемента в заданную позицию ArrayList. Обратите внимание на вызов метода ToArray(), который возвращает общий массив типов System.Object на основе содержимого оригинального ArrayList. На рис. 7.13 показан соответствующий вывод.

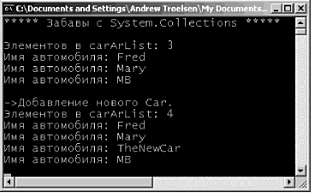


Рис. 7.13. Забавы с System.Collections.ArrayList

**Работа с типом Queue**

Тип Queue (очередь) – это контейнер, гарантирующий размещение элементов по правилу "первым прибыл – первым обслужен". К сожалению, люди сталкиваются с очередями повсеместно: очереди в банке, кинотеатре, по утрам к автомату, продающему кофе, и т.д. При моделировании сценариев, в которых элементы обрабатываются по правилу очереди, на помощь приходит System.Collections.Queue. Вдобавок к функциональным возможностям, обеспечиваемым поддерживаемыми интерфейсами, Queue определяет ряд членов, описанных в табл. 7.4.

**Таблица 7.4.**Члены типа Queue

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Dequeue() | Возвращает объект, находящийся в начале Queue, с одновременным его удалением |
| Enqueue() | Добавляет объект в конец Queue |
| Peek() | Возвращает объект, находящийся в начале Queue, без его удаления |

Чтобы проиллюстрировать возможности этих методов, снова используем нашу автомобильную тему и построим объект Queue, моделирующий очередь автомобилей перед въездом на мойку. Во-первых, предположим, что у нас есть следующий вспомогательный статический метод.

public static void WashCar(Car с) {

 Console.WriteLine("Моется {0}", с.petName);

}

Теперь рассмотрим следующий программный код.

static void Main(string[] args) {

 …

**// Создание очереди с тремя элементами.**

 Queue carWashQ = new Queue();

 carWashQ.Enqueue(new Car ("Первая", 0, 1));

 carWashQ.Enqueue(new Car("Вторая", 0, 2));

 carWashQ.Enqueue(new Car("Третья"*,*0, 3));

**// Первая машина в очереди.**

 Console.WriteLine("Первой в очереди является {0}", ((Сar)сarWashQ.Peek()).petName);

**// Удаление всех элементов из очереди.**

 WashCar((Car)carWashQ.Dequeue());

 WashCar((Car)carWashQ.Dequeue());

 WashCar((Car)carWashQ.Dequeue());

**// Попытаемся удалить снова?**

 try {WashCar((Car)carWashQ.Dequeue());}

 catch(Exception е) { Console.WriteLine("Ошибка: {0}", e.Message);}

}

Здесь в тип Queue с помощью метода Enqueue() вставляются три элемента. Вызов Реек() позволяет проверить, (но не удалить) первый элемент в текущем состоянии Queue, и таким элементом в данном случае является машина с именем Первая. Наконец, с помощью Dequeue() элемент из очереди удаляется и посылается во вспомогательную функцию WashСar() для обработки. Обратите внимание на то, что при попытке удаления элемента из пустой очереди среда выполнения генерирует исключение.

**Работа с типом Stack**

Тип System.Collections.Stack представляет коллекцию, в которой элементы размещаются по правилу "последним прибыл – первым обслужен". Как и следует ожидать, Stack определяет члены с именами Push() и Pop() (для добавления элементов в стек и удаления их из стека). В следующем примере стека используется стандартный тип System.String.

static void Main(string[] args) {

 …

 Stack stringStack = new Stack();

 stringStack.Push("Первый");

 stringStack.Push("Второй");

 stringStack.Push("Третий");

 // **Смотрим на первый элемент, удаляем его и смотрим снова.**

 Console.WriteLine("Первый элемент: {0}", stringStack.Peek());

 Console.WriteLine("Удален {0}", stringStack.Pop());

 Console.WriteLine("Первый элемент: {0}", stringStack.Peek());

 Console.WriteLine("Удален {0}", stringStack.Pop());

 Console.WriteLine("Первый элемент: {0}", stringStack.Peek());

 Console.WriteLine("Удален {0}", stringStack.Pop());

 try {

  Console.WriteLine("Первый элемент: {0}", stringStack.Peek());

  Console.WriteLine ("Удален {0}", stringStack.Pop());

 } catch(Exception e) {Console.WriteLine("Ошибка: {0}", e.Message);}

}

Здесь строится стек, содержащий три строковых типа (названных в соответствии с порядком их вставки). "Заглядывая" в стек, вы видите элемент, находящийся на вершине стека, поэтому первый вызов Peek() выявляет третью строку. После серии вызовов Pop() и Peek() стек, в конечном счете, опустошается, и тогда дополнительный вызов Peek()/Pop() приводит к генерированию системного исключения.

**Исходный код.**Проект CollectionTypes размещен в подкаталоге, соответствующем главе 7.

**Пространство имен System.Collections.Specialized**

Кроме типов, определенных в пространстве имен System.Collections, библиотеки базовых классов .NET предлагают набор более специализированных типов, определенных в пространстве имен System.Collections.Specialized. Например, типы StringDictionary и ListDictionary обеспечивают "стилизованную" реализацию интерфейса IDictionary. Описания основных типов класса из этого пространства имен предлагаются в табл. 7.5.

**Таблица 7.5.**Типы пространства имен System.Collections.Specialized.

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| CollectionsUtil | Создает коллекции, игнорирующие регистр символов в строках |
| HybridDictionary | Реализует IDictionary, используя ListDictionary, пока коллекция мала, и переключаясь на Hashtable, когда коллекция становится большой |
| ListDictionary | Реализует IDictionary, используя однонаправленный список. Рекомендуется для коллекций, которые содержат не более десятка элементов |
| NameValueCollection | Представляет отсортированную коллекцию связанных ключей и значений типа String, которые могут быть доступны или по ключу, или по индексу |
| StringCollection | Представляет коллекцию строк |
| StringDictionary | Реализует Hashtable с ключом, строго типизированным, как строка, а не объект |
| StringEnumerator | Поддерживает простой цикл по элементам StringCollection |

**Резюме**

Интерфейс можно определить, как именованную коллекцию *абстрактных членов.*Ввиду того, что интерфейс не предлагает деталей реализаций, он обычно рассматривается, как, вариант поведения, возможного для данного типа. При реализации одного и того же интерфейса в нескольких классах вы получаете возможность обращаться с соответствующими типами одинаково (это называется интерфейсным полиморфизмом).

Для определения новых интерфейсов в C# предлагается ключевое слово interface. Любой тип может поддерживать столько интерфейсов, сколько необходимо, нужно только указать их в списке определения типа, разделяя запятыми. При этом можно создавать интерфейсы, которые оказываются производными нескольких базовых интерфейсов.

Помимо возможности создания пользовательских интерфейсов, вы имеете возможность использовать ряд стандартных интерфейсов, определенных в библиотеках .NET (т.е. предлагаемых каркасом приложений). Вы можете строить пользовательские типы, реализующие эти встроенные интерфейсы, чтобы обеспечить выполнение ряда стандартных действий, таких как клонирование, сортировка или перечисление. Наконец, в этой главе вы узнали о классах коллекций, определенных в пространстве имен System.Collections, и изучили ряд общих интерфейсов, которые используются связанными с коллекциями типами.

**ГЛАВА 8. Интерфейсы обратного вызова, делегаты и события**

До этого момента в нашей книге в каждом примере приложении программный код Main() тем или иным способом направлял запросы соответствующим объектам. Но мы пока что не рассматривали возможность обратного обращения объекта к вызывающей стороне. В большинстве программ типичным для объектов системы оказывается двухсторонняя коммуникация, которая реализуется с помощью интерфейсов обратного вызова, событий и других программных конструкций. Эта глава начинается с рассмотрения того, как с помощью типов интерфейса реализовать функциональные возможности обратного вызова.

Затем вы узнаете о типе делегата .NET, который является объектом, обеспечивающим типовую безопасность и "указывающим" на метод или методы, которые могут быть вызваны позднее. Но, в отличие от традиционного указателя на функцию в C++, делегаты .NET представляют собой объекты, которые имеют встроенную поддержку многоадресного и асинхронного вызова методов. Мы рассмотрим асинхронное поведение типов делегата позже, при изучении пространства имен System.Threading (см. главу 14).

Выяснив, как создавать типы делегата и работать с ними, мы с вами рассмотрим ключевое слово C# event, которое призвано упростить и ускорить работу с типами делегата. Наконец, в этой главе будут рассмотрены новые возможности языка C#*,*связанные с использованием делегатов и событий, включая анонимные методы и групповые преобразования методов, Вы увидите, что указанный подход открывает более короткий путь к целевым объектам соответствующих событий.

**Интерфейсы обратного вызова**

При рассмотрении материала предыдущей главы вы имели возможность убедиться, что интерфейсы определяют доведение, которое может поддерживаться самыми разными типами. Кроме использований интерфейсов для поддержки полиморфизма, их можно использовать и в качестве *механизма обратного вызова.*Соответствующий подход дает объектам возможность осуществлять двухсторонний обмен, используя общее множество членов.

Чтобы показать варианты использования интерфейсов обратного вызова, мы изменим уже знакомый нам тип Car так, чтобы он мог информировать вызывающую сторону о приближении поломки машины (т.е. о том, что текущая скорость на 10 км/ч ниже максимальной скорости) и о свершившейся поломке (когда текущая скорость равна или выше максимальной скорости). Способность посылать и принимать соответствующие события будет реализована с помощью интерфейса, носящего имя IEngineEvents.

**// Интерфейс обратного вызова.**

public interface IEngineEvents {

 void AboutToBlow(string msg);

 void Exploded (string msg);

}

Интерфейсы событий обычно реализуются не объектом, непосредственно "заинтересованным" в получении событий, а некоторым вспомогательным объектом, который называется *объектом-приемником.*Отправитель событий (в данном случае это тип Car) при определенных условиях выполняет для приемника соответствующие вызовы. Предположим, что класс приемника называется CarEventSink, и он просто выводит поступающие сообщения на консоль. Кроме того, наш приемник содержит строку, в которой указано его информативное имя.

**// Приемник событий Car.**

public class CarEventSink: IEngineEvents {

 private string name;

 public CarEventSink(){}

 public CarEventSink(string sinkName) { name = sinkName; }

 public void AboutToBlow(string msg) { Console.WriteLine("{0} сообщает: {1}", name, msg); }

 public void Exploded(string msg) { Console.WriteLine(" {0} сообщает: {1}", name, msg); }

}

Теперь, когда у нас есть объект-приемник, реализующий интерфейс событий, нашей следующей задачей является передача ссылки на этот приемник в тип Car. Тип Car будет хранить эту ссылку и при необходимости выполнять обратные вызовы приемника. Чтобы тип Car мог получить ссылку на приемник, нужно добавить в тип Car вспомогательный член, который мы назовем Advise(). Точно так же. если вызывающая сторона пожелает отменить привязку к источнику событий, она может вызвать другой вспомогательный метод типа Car – метод Unadvise(). Наконец, чтобы позволить вызывающей стороне регистрировать множество приемников событий (с целью групповой адресации), тип Car поддерживает ArrayList для представления исходящих соединений.

**// Тип Car и вызывающая сторона могут связываться**

**// с помощью интерфейса IEngineEvents.**

public class Car {

**// Набор связанных приемников.**

 ArrayList clientSinks = new ArrayList();

**// Присоединение к источнику событий или отсоединение от него.**

 public void Advise(IEngineEvents sink) {clientSinks.Add(sink);}

 public void Unadvise(IEngineEvents sink) {clientSinks.Remove(sink);}

 …

}

Чтобы на самом деде посылать события, мы обновим метод Car.Accelerate() так, чтобы он осуществлял "проход" по соединениям, указанным в ArrayList, и при необходимости выдавал подходящее сообщение (обратите внимание на то, что теперь в классе Car есть член-переменная carIsDead логического типа для представления состояния двигателя машины).

**// Протокол событий на базе интерфейса.**

class Car {

 …

**// Эта машина работает или нет?**

 bool carIsDead;

 public void Accelerate(int delta) {

**// Если машина 'сломалась', отправить событие Exploded**

**// каждому приемнику.**

  if (carIsDead) {

   foreach(IEngineEvents e in clientSinks) e.Exploded("Извините, машина сломалась…");

  } else {

   currSpeed += delta;

**// Отправка события AboutToBlow.**

   if (10 == maxSpeed – currSpeed) {

    foreach(IEngineEvents e in clientSinks) е.AboutToBlow("Осторожно! Могу сломаться!");

   }

   if (currSpeed ›= maxSpeed) carIsDead = true;

   else Console.WriteLine(" \tCurrSpeed = {0} ", currSpeed);

  }

 }

}

Вот подходящий программный код клиента, в котором используется интерфейс обратного вызова для отслеживания событий Car.

**// Создание машины и мониторинг событий.**

public class CarApp {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\* Интерфейсы и контроль событий \*\*\*");

  Car cl = new Car("SlugBug", 100, 10);

**// Создание объекта-приемника.**

  CarEventSink sink = new CarEventSink();

**// Передача Car ссылки на приемник.**

  cl.Advise(sink);

**// Ускорение (вызывает наступление событий).**

  for (int i = 0; i ‹ 10; i++) cl.Accelerate(20);

**// Разрыв связи с источником событий.**

  cl.Unadvise(sink);

  Console.ReadLine();

 }

}

На рис. 8.1 показан конечный результат работы этого основанного на интерфейсе протокола событий.

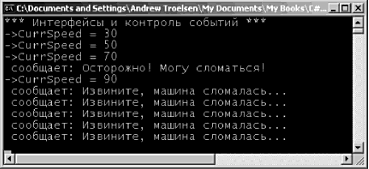


Рис. 8.1. Основанный на интерфейсе протокол событий

Обратите внимание на то, что метод Unadvise() позволяет вызывающей стороне селективно отключаться от источника событий. Здесь перед выходом из Main() вызывается Unadvise(), хотя, строго говоря, это и не обязательно. Но предположим, что приложение должно зарегистрировать два приемника, динамически отключить их по очереди в процессе выполнения, а затем продолжить работу.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Интерфейсы и контроль событий \*\*\*\*\*");

 Car cl = new Car("SlugBug", 100, 10);

**// Создание двух объектов.**

 Console.WriteLine("\*\*\*\*\* Создание приемников \*\*\*\*\*");

 CarEventSink sink = new CarEventSink("Первый приемник");

 CarEventSink myOtherSink = new CarEventSink("Второй приемник");

**// Передача приемников объекту Car.**

 Console.WriteLine("\n\*\*\*\*\* Отправка приемников в Car \*\*\*\*\*");

 cl.Advise(sink);

 cl.Advise(myOtherSink);

**// Ускорение (при этом генерируются события).**

 Console.WriteLine("\n\*\*\*\*\* Ускорение \*\*\*\*\*");

 for (int i = 0; i ‹ 10; i++) cl.Accelerate(20);

**// Отключение первого приемника событий.**

 Console.WriteLine("\n\*\*\*\*\* Отключение первого приемника \*\*\*\*\*");

 cl.Unadvise(sink);

**// Новое ускорение (теперь вызывается только myOtherSink)**.

 Console.WriteLine("\n\*\*\*\*\* Снова ускорение \*\*\*\*\*);

 for(int i = 0; i ‹ 10; i++) cl.Accelerate(20);

**// Отключение второго приемника событий.**

 Console.WriteLine("\n\*\*\*\*\* Отключение второго приемника \*\*\*\*\*");

 Console.ReadLine();

}

Интерфейсы событий могут быть полезны и тем*,*что они могут использоваться с любыми языками и любыми платформами (.NET, J2EE или какими-то иными), поддерживающими программирование на основе интерфейсов. Однако "официальный" протокол событий задает платформа .NET. Чтобы понять внутреннюю архитектуру обработки событий, мы начнем с обсуждения роли типа делегата.

**Исходный код**. Проект EventInterface размещен в подкаталоге, соответствующем главе 8.

**Тип делегата .NET**

Перед тем как дать формальное определение делегата .NET, давайте обсудим соответствующие перспективы. В Windows API для создания объектов, называемых *функциями обратного вызова,*предполагается использовать указатели функций (подобные указателям C). Используя обратный вызов, программисты могут создавать функции, возвращающие информацию другим функциям в приложении в ответ на их вызов.

Проблема стандартных функций обратного вызова, подобных C, заключается в том, что такие функции, по сути, мало отличаются от простой ссылки на адрес в памяти. В идеале функции обратного вызова могли бы содержать дополнительную обеспечивающую безопасность информацию о числе (и типах) параметров и возвращаемом значении для соответствующего метода. К сожалению, для традиционных функций обратного вызова это не так, и, как вы можете догадаться, (именно это часто оказывается причиной дефектов, которые проявляются в среде выполнения и которые трудно устранить.

Тем не менее, возможность обратного вызова является полезной. В .NET Framework обратный вызов поддерживается» и соответствующие функциональные возможности реализуются с помощью более безопасных методов делегата, что обеспечивает более совершенный объектно-ориентированный подход. В сущности, *делегат*- это обеспечивающий типовую безопасность объект, указывающий на другой метод или, возможно, несколько методов в приложении, которые могут быть вызваны с помощью делегата позже. Точнее, тип делегата хранит три следующих элемента информации:

• *имя*метода, к которому должен обращаться вызов;

• *аргументы*метода (если таковые имеются);

• *возвращаемое значение*метода (если таковое предполагается).

**Замечание.**В отличие от указателей функций C(++), делегаты .NET могут указывать на статические методы и на методы экземпляра.

После создания делегата и получения вышеуказанной информации делегат может динамически в среде выполнения вызывать методы, на которые он указывает. Вы убедитесь, что в .NET Framework каждый делегат .NET (в том числе и ваши пользовательские делегаты) автоматически наделяется способностью вызывать свои методы *синхронно*или *асинхронно.*Это очень упрощает задачи программирования, поскольку позволяет вызвать метод во вторичном потоке выполнения без явного создания объекта Thread и управления им вручную. Мы рассмотрим асинхронное поведение типов делегата в ходе нашего исследования пространства имен System.Threading в главе 14.

**Определение делегата в C#**

Чтобы создать делегат в C#, вы должны использовать ключевое слово delegate. Имя делегата может быть любым. Однако делегат должен соответствовать методу, на который этот делегат будет указывать. Предположим, например, что нам нужно создать делегат с именем BinaryOp, который сможет указывать на любой метод, возвращающий целое число и имеющий целочисленные входные параметры.

**// Этот делегат может указывать на любой метод,**

**// принимающий два целых значения**

**// и возвращающий целое значение.**

public delegate int BinaryOp(int x, int y);

При обработке типов делегата компилятор C# автоматически генерирует изолированный класс, являющийся производным от System.MulticastDelegate. Этот класс (вместе с базовым классом System.Delegate) обеспечивает делегату необходимую инфраструктуру, позволяющую поддерживать список методов, которые должны быть вызваны позднее. Например, если рассмотреть содержимое делегата BinaryOp с помощью ildasm.exe, вы увидите элементы, показанные на рис. 8.2.

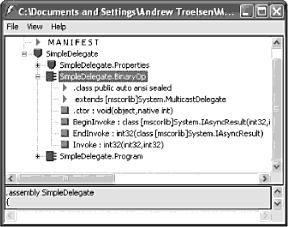


Рис. 8.2. Ключевое слово delegate в C# представляет изолированный тип, производный от System.MulticastDelegate

Как видите, генерируемый здесь класс BinaryOp определяет три открытых метода. Метод Invoke() можно назвать главным, поскольку он используется для *синхронного*вызова методов, поддерживаемых типом делегата, и синхронность здесь означает то, что вызывающая сторона для продолжения работы должна ожидать завершения вызова. Весьма странным кажется тот факт, что синхронный метод Invoke() в C# нельзя вызвать непосредственно. Чуть позже будет продемонстрировано, как Invoke() вызывается опосредованно с помощью соответствующей синтаксической конструкции.

Методы BeginInvoke() и EndInvoke() обеспечивает возможность *асинхронного*вызова текущего метода во вторичном потоке выполнения. Если у вас есть опыт работы с многопоточными приложениями, вы должны знать, что одной из главных причин, по которым разработчики создают вторичные потоки, является вызов методов, для выполнения которых требуется много времени. И хотя библиотеки базовых классов .NET предлагают целое пространство имен (System.Threading), специально предназначенное для решения задач многопоточного программирования, с помощью делегатов соответствующие функциональные возможности использовать проще.

Но откуда компилятор "знает", как определять методы Invoke(), BeginInvoke() и EndInvoke()? Чтобы понять суть процесса, рассмотрим пример автоматически генерируемого типа класса BinаrуОр (полужирным шрифтом здесь обозначены элементы, заданные определяемым типом делегата).

sealed class **BinaryOp**: System.MulticastDelegate {

 public **BinaryOp**(object target, uint functionAddress);

 public **void** Invoke(int x, int y);

 public IAsyncResult BeginInvoke(**int x, int y,**AsyncCallback cb, object state);

 public **int** EndInvoke(IAsyncResult result);

}

Во-первых, обратите внимание на то, что параметры и возвращаемое значение определяемого здесь метода Invoke() соответствуют определению делегата BinaryOp. Первые параметры членов BeginInvoke() (в данном случае это два целых числа) тоже соответствуют определению делегата BinaryOp, однако BeginInvoke() всегда имеет еще два параметра (типа AsyncCallback и object), которые используются для асинхронного вызова методов. Наконец, возвращаемое значение метода EndInvoke() тоже соответствует исходной декларации делегата, а единственным параметром метода является объект, реализующий интерфейс IAsyncResult.

Рассмотрим еще один пример. Предположим, что мы определили тип делегата, который позволяет указать на любой метод, возвращающий строку и имеющий три входных параметра System.Boolean.

public delegate string MyDelegate(bool a, bool b, bool c);

На этот раз автоматически генерируемый класс выглядит так.

sealed class **MyDelegate**: System.MulticastDelegate {

 public **MyDelegate**(object target, uint functionAddress);

 public **string**Invoke(**bool a, bool b, bool c**);

 public IAsyncResult BeginInvoke(**bool a, bool b, bool c,**AsyncCallback cb, object state);

 public **string**Endlnvoke(IAsyncResult result);

}

Делегаты могут также "указывать" на методы, содержащие любое число параметров out или ref. В качестве примера рассмотрим следующий тип делегата.

public delegate string MyOtherDelegate**(out**bool a, **ref**bool b, int c);

Сигнатуры методов Invoke() и BeginInvoke() выглядят так, как и ожидается, но обратите внимание на метод EndInvoke(), который теперь включает и все аргументы out/ref, определенные типом делегата.

sealed class **MyOtherDelegate**: System.MulticastDelegate {

 public **MyOtherDelegate**(object target, uint functionAddress);

 public **string**Invoke(**out**bool a, **ref**bool b, int c);

 public IAsyncResult BeginInvoke(**out**bool a, **ref**bool b, int c, AsyncCallback cb, object state);

 public **string**EndInvoke**(out**bool a, **ref**bool b, IAsyncResult result);

}

Итак, в результате определения делегата в C# компилятор генерирует изолированный класс с тремя методами, для которых возвращаемые значения и типы параметров соответствуют декларации делегата. Следующий псевдокод приближенно описывает соответствующий базовый шаблон.

**// Это только псевдокод!**

public sealed class ***ИмяДелегата****:*System.MulticastDelegate {

 public ***ИмяДелегата***(object target, uint functionAddress);

 public ***возвращаемоеЗначениеДелегата***Invoke(***всеПараметрыДелегата***);

 public IAsyncResult BeginInvoke(***всеПараметрыДелегата****,*AsyncCallback cb, object state);

 public ***возвращаемоеЗначениеДелегата***EndInvoke(всеПараметрыRefOutДелегата, IAsyncResult result);

}

**Базовые классы System.MulticastDelegate и System.Delegate**

Таким образом, при создании типов c помощью) ключевого слова delegate в C# вы неявно объявляете тип класса, являющегося производным от System.MulticastDelegate. Этот класс обеспечивает своим потомкам доступ к списку с адресами тех методов, которые поддерживаются типом делегата, а также предлагает несколько дополнительных методов (и ряд перегруженных операций), обеспечивающих взаимодействие со списком вызовов. Вот программный код некоторых членов System.MulticastDelegate.

[Serializable]

public abstract class MulticastDelegate: Delegate {

**// Методы**

public sealed override Delegate[] GetInvocationList();

 public static bool operator==(MulticastDelegate d1, MulticastDelegate d2);

 public static bool operator!=(MulticastDelegate d1, MulticastDelegate d2);

**// Поля**

 private IntPtr \_invocationCount;

 private object \_invocationList;

}

Дополнительные функциональные возможности System.MulticastDelegate получает от своего родительского класса System.Delegate. Вот часть определения этого класса.

[Serializable, ClassInterface(ClassInterfaceType.AutoDual)]

public abstract class Delegate: ICloneable, ISerializable {

**// Методы**

 public static Delegate Combine(params Delegate[] delegates);

 public static Delegate Combine(Delegate a, Delegate b);

 public static Delegate Remove(Delegate source, Delegate value);

 public static Delegate RemoveAll(Delegate source, Delegate value);

**// Перегруженные операции**

 public static bool operator==(Delegate d1, Delegate d2);

 public static bool operator!=(Delegate d1, Delegate d2);

**// Свойства**

 public MethodInfo Method {get;}

 public object Target {get;}

}

Здесь следует подчеркнуть, что от вас никогда не потребуется непосредственно получать производные этих базовых классов, и вы можете ограничить себя использованием только членов, указанных в табл. 8.1.

**Таблица 8.1.**Избранные члены System.MulticastDelegate и System.Delegate

|  |  |
| --- | --- |
| **Наследуемый член** | **Описание** |
| Method | Свойство, возвращающее тип System.Reflection.Methodlnfo, который представляет информацию о статическом методе, поддерживаемом делегатом |
| Target | Если доступный для вызова метод определен на уровне объекта (а не как статический метод), то Target возвратит имя метода, поддерживаемого делегатом. Если возвращенным значением Target оказывается null, то доступный для вызова метод является статическим |
| Combine() | Статический метод, добавляющий метод в список методов, поддерживаемых делегатом. В C# этот метод вызывается с помощью перегруженной операции += |
| GetInvocationList() | Метод, возвращающий массив типов System.Delegate, каждый из которых представляет конкретный доступный для вызова метод |
| Remove() RemoveAll() | Статические методы, удаляющие метод (или все методы) из списка вызовов. В C# метод Remove() можно вызвать опосредованно, используя перегруженную операцию –= |

**Простейший пример делегата**

В начале освоения приемов работы с делегатами у программиста может возникать много вопросов. Поэтому мы начнем с рассмотрения очень простого примера, в котором используется созданный нами тип делегата BinaryOp. Вот программный код, который мы собираемся проанализировать.

namespace SimpleDelegate {

**// Этот делегат может указывать на любой метод,**

**// принимавший два целых значения**

**// и возвращающий целое значение.**

 public delegate int BinaryOp(int x, int y);

**// Этот класс содержит методы, на которые**

**// будет указывать BinaryOp.**

 public class SimpleMath {

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

  public static int Subtract(int x, int y) {return x – у;}

 }

 class Program {

  static void Main(string[] args) {

   Console.WriteLine("\*\*\*\*\* пример делегата \*\*\*\*\*\n");

**// Создание объекта BinaryOp,**

**// "указывающего" на SimpleMath.Add().**

   BinaryOp b = new BinaryOp(SimpleMath.Add);

**// Вызов метода Add() с помощью делегата.**

   Console.WriteLine(''10 + 10 равно {0}", b(10, 10));

   Console.ReadLine();

  }

 }

}

Снова обратите внимание на формат делегата BinaryOp, который может указывать на любой метод, принимающий два целых значения и возвращающий целое значение. Итак, мы создали класс с именем SimpleMath, определяющий два статических метода, которые (как неожиданно) соответствуют шаблону, определенному делегатом BinaryOp.

Чтобы добавить целевой метод в делегат, нужно просто передать имя этою метода конструктору делегата. Тогда вы сможете вызвать указанный член с помощью синтаксической конструкции, подобной прямому вызову функции.

**// Здесь на самом деле вызывается Invoke()**!

Console.WriteLine("10 + 10 is {0}", b(10, 10));

"За кулисами" среда выполнения вызывает сгенерированный компилятором метод Invoke(). Вы можете проверить это сами, если откроете компоновочный блок с помощью ildasm.exe и посмотрите на программный код CIL метода Main().

.method private hidebysig static void Main(string[] args) cil managed {

 …

 .locals init ([0] class SimpleDelegate.BinaryOp b)

 ldftn int32 SimpleDelegate.SimpleMath::Add(int32, int32)

 …

 newobj instance void SimpleDelegate.BinaryOp::.ctor(object, native int)

 stloc.0

 ldstr "10 + 10 is {0}"

 ldloc.0

 ldc.i4.s 10

 ldc.i4.s 10

 callvirt instance int32 **SimpleDelegate.BinaryOp::Invoke**(int32, int32)

 …

}

Напомним, что делегаты .NET (в отличие от указателей функций в C) обеспечивают типовую безопасность. Поэтому, если вы попытаетесь передать делегату метод, "не соответствующий шаблону", вы получите сообщение об ошибке компиляции. Например, предположим, что класс SimpleMath определяет еще один метод, носящий имя SquareNumber().

public class SimpleMath {

 public static int SquareNumber(int a) { return a \* a; }

}

С учетом того, что делегат BinaryOp может указывать только на методы, принимающие два целых значения и возвращающие целое значение, следующий программный метод оказывается некорректным и компилироваться не будет.

**// Ошибка! Метод не соответствует шаблону делегата!**

BinaryOp b = new BinaryOp(SimpleMath.SquareNumber);

**Исследование объекта делегата**

Добавим в имеющийся пример вспомогательную функцию с именем DisplayDelegateInfo(). Она будет выводить имена методов, поддерживаемых поступающим типом, производным от System.Delegate, а также имя класса, определяющего метод. Для этого мы выполним цикл по элементам массива System.Delegate, возвращенного из GetInvocationList(), вызывая свойства Target и Method для каждого объекта.

static void DisplayDelegateInfo(Delegate delObj) {

**// Вывод имен каждого из элементов**

**// списка вызовов делегата.**

 foreach (Delegate d in delQbj.GetInvocationList()) {

  Console.WriteLine("Имя метода: {0}", d.**Method**);

  Console.WriteLine("Имя типа: {0}", d.**Target**);

 }

}

Если изменить метод Main() так, чтобы он вызывал этот новый вспомогательный метод, то вы увидите вывод, показанный на рис. 8.3.



Рис. 8.3. Проверка списка вызовов делегата

Обратите внимание на то, что здесь имя типа (SimpleMath) свойством Target не отображается. Причина в том, что наш делегат BinaryOp указывает на *статические*методы, следовательно, нет объекта, на который нужно ссылаться! Но если изменить методы Add() и Subtract() так, чтобы они перестали быть статическими, можно создать экземпляр типа SimpleMath и указать методы для вызова так, как показано ниже.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Пример делегата \*\*\*\*\*\n");

**// Делегаты .NET могут указывать на методы экземпляра.**

 SimpleMath m = new SimpleMath();

 BinaryOp b = new BinaryOp(m.Add);

**// Вывод информации об объекте.**

 DisplayDelegateInfо(b);

 Console.WriteLine("\n10 + 10 равно {0}", b(10, 10));

 Console.ReadLine();

}

Теперь вы должны увидеть вывод, показанный на рис. 8.4.



Рис. 8.4. Проверка списка вызовов делегата (новая попытка)

**Исходный код**. Проект SimpleDelegate размещен в подкаталоге, соответствующем главе 8.

**Модификация типа Car с учетом делегатов**

Очевидно, что предыдущий пример SimpleDelegate был исключительно иллюстративным, поскольку нет никаких реальных причин строить делегаты для простого сложения двух чисел. Но этот пример раскрывает принципы работы с типами делегата. Для построения более реального примера мы модифицируем тип Car так, чтобы он посылал сообщения Exploded и AboutToBlow через делегаты .NET, a не через пользовательский интерфейс обратного вызова. Кроме отказа от реализации IEngineEvents, мы должны выполнить следующие шаги:

• определить делегаты AboutToBlow и Exploded;

• объявить члены-переменные всех типов делегата в классе Car;

• создать вспомогательные функции Car, которые позволят вызывающей стороне указать методы, поддерживаемые членами-переменными делегатов;

• обновить метод Accelerate(), чтобы иметь возможность в подходящей ситуации обратиться к списку вызовов делегата.

Рассмотрите следующий обновленный класс Car, в котором решены первые три из указанных задач.

public class Car {

**// Определение типов делегата.**

 public delegate void AboutToBlow(string msg);

 public delegate void Exploded(string msg);

**// Определение членов-переменных для каждого из типов.**

 private AboutToBlow almostDeadList;

 private Exploded explodedList;

**// Добавление элементов в список вызовов**

**// с помощью вспомогательных методов.**

 public void OnAboutToBlow(AboutToBlow clientMethod) {almostDeadList = clientMethod;}

 public void OnExploded(Exploded clientMethod) {explodedList = clientMethod;}

 …

}

Обратите внимание на то, что в этом примере мы определяем типы делегата непосредственно в рамках типа Car. Если исследовать библиотеки базовых классов, то станет ясно, что определение делегата в рамках типа, с которым он обычно работает, является вполне типичным. В связи с этим, поскольку компилятор преобразует делегат в полное определение класса, мы здесь фактически создаем вложенные классы.

Далее обратите внимание на то, что здесь объявлены члены-переменные (по одному для каждого типа делегата) и вспомогательные функции (OnAboutToBlow() и OnExploded()), которые позволят клиенту добавлять методы в списки вызовов делегатов. В принципе эти методы подобны методам Advise() и Unadvise(), которые были нами созданы в примере с EventInterfасе. Но в данном случае входящим параметром оказывается размещаемый клиентом объект делегата, а не класс, реализующий конкретный интерфейс.

Здесь мы должны обновить метод Accelerate(), чтобы вызывались делегаты, а не просматривались объекты ArrayList приемников клиента (как это было в примере с EventInterfасе). Подходящая модификация может выглядеть так.

public void Accelerate(int delta) {

**// Если машина 'сломалась', генерируется событие Exploded.**

 if (carIsDead) {

  if (explodedList != null) explodedList("Извините, машина сломалась…");

 } elsе {

  currSpeed += delta;

**// Вот-вот сломается?**

  if (10 == maxSpeed – currSpeed && almostDeadList != null) {

   almostDeadList("Осторожно! Могу сломаться!");

  }

**// Пока все OK!**

  if (currSpeed ›= maxSpeed) carIsDead = true;

  else Console.WriteLine("CurrSpeed = {0}", currSpeed);

 }

}

Обратите внимание на то, что перед вызовом методов, связанных с членами-переменными almostDeadList и explodedList, их значения проверяются на допустимость. Причина в том, что размещение соответствующих объектов с помощью вызова вспомогательных методов OnAboutToBlow() и OnExploded() будет задачей вызывающей стороны. Если вызывающая сторона не вызовет эти методы, а мы попытаемся получить список вызовов делегата, то будет сгенерировано исключение NullReferenseException и в среде выполнения возникнут проблемы (что, конечно же, нежелательно).

Теперь, когда инфраструктура делегата имеет нужный нам вид, рассмотрим модификацию класса Program.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Делегаты и контроль событий \*\*\*\*\*");

**// Обычное создание класса Car*.***

  Car cl = new Car("SlugBug", 100, 10);

**// Регистрация обработчиков событий для типа Car.**

  cl.OnAboutToBlow(new Car.AboutToBlow(CarAboutToBlow));

  cl.OnExploded(new Car.Exploded(CarExploded));

**// Ускоряемся (при этом генерируются события) .**

  Console.WriteLine("\n\*\*\*\*\* Ускорение \*\*\*\*\*");

  for(int i = 0; i ‹ 6; i++) cl.Accelerate(20);

  Console.ReadLine();

 }

**// Car будет вызывать эти методы.**

 public static void CarAboutToBlow(string msg) {Console.WriteLine(msg);}

 public static void CarExploded(string msg) {Console.WriteLine(msg);}

}

Здесь следует отметить только то, вызывающая сторона задает значения членам-переменным делегата с помощью вспомогательных методов регистрации. Кроме того, поскольку делегаты AboutToBlow и Exploded вложены в класс Car, при их размещении следует использовать полные имена (например, Car.AboutToBlow). Как любому конструктору, мы передаем конструктору делегата имя метода, который нужно добавить в список вызовов. В данном случае это два статических члена класса Program (если вложить указанные методы в новый класс, это будет очень похоже на тип CarEventSink из примера Event Interface).

**Реализация групповых вызовов**

Напомним, что делегаты .NET наделены возможностью *группового вызова.*Другими словами, объект делегата может поддерживать не один метод, а целый список доступных для вызова методов. Когда требуется добавить в объект делегата несколько методов, используется перегруженная операция +=, а не прямое присваивание. Чтобы разрешить групповой вызов для типа Car, можно обновить методы OnAboutToBlow() и OnExploded() так, как показано ниже.

public class Car {

**// Добавление элемента в список вызовов.**

 public void OnAboutToBlow(AboutToBlow clientMethod) {almostDeadList += clientMethod;}

 public void OnExploded(Exploded clientMethod) {explodedList += clientMethod;}

 …

}

Теперь вызывающая сторона может зарегистрировать несколько целевых объектов.

class Program {

 static void Main(string[] args) {

  Car c1 = new Car("SlugBug", 100, 10);

**// Регистрация множества обработчиков событий.**

  c1.OnAboutToBlow(new Car.AboutToBlow(CarAboutToBlow));

  c1.OnAboutToBlow(new Car.AboutToBlow(CarlsAlmostDoomed));

  c1.OnExploded(new Car.Exploded(CarExploded));

  …

 }

**// Car будет вызывать эти методы.**

 public static void CarAboutToBlow(string msg) {Console.WriteLine (msg);}

 public static void CarIsAlmostDoomed(string msg) {Console.WriteLine("Важное сообщение от Car: {0}", msg);}

 public static void CarExploded(string msg) {Console.WriteLine(msg);}

}

В программном воде CIL операция += преобразуется в вызов статического метода Delegate.Combine() (можно было бы вызвать Delegate.Combine() непосредственно, но операция += предлагает более простую альтернативу). Взгляните, например, на CIL-представление метода OnAboutToBlow().

.method public hidebysig instance void **OnAboutToBlow**(class CarDelegate.Car/AboutToBlow clientMethod) cil managed {

 .maxstack 8

 ldarg.0

 dup

 ldfld class CarDelegate.Car/AboutToBlow CarDelegate.Car::almostDeadList

 ldarg.1

**call class [mscorlib]System.Delegate [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)**

 castclass CarDelegate.Car/AboutToBlow

 stfld class СarDelegate.Car/AboutToBlow CarDelegate.Car::almostDeadList

 ret

}

Класс Delegate определяет также статический метод Remove(), который позволит вызывающей стороне динамически удалять элементы из списка вызовов. Легко догадаться, что в C# разработчики могут для этого использовать перегруженную операцию -=. Чтобы предоставить вызывающей стороне возможность не привязываться к обозначениям AboutToBlow и Exploded, можно добавить в тип Car следующие вспомогательные методы (обратите внимание на операцию -=).

public class Car {

**// Удаление элемента из списка вызовов.**

 public void RemoveAboutToBlow(AboutToBlow clientMethod) {almostDeadList -= clientMethod;}

 public void RemoveExploded(Exploded clientMethod) {explodedList -= clientMethod;}

 ...

}

Здесь синтаксис -= тоже выступает в качестве простого сокращения для вызова статического метода Delegate.Remove(), что доказывается следующим программным кодом CIL для члена RemoveAboutToBlow() типа Car.

.method public hidebysig instance void RemoveAboutToBlow(class CarDelegate.Car/AboutToBlow clientMethod) cil managed {

 .maxstack 8

 ldarg.0

 dup

 ldfld class CarDelegate.Car/AboutToBlow CarDelegate.Car::almostDeadList

 ldarg.1

**call class [mscorlib]System.Delegate**[mscorlib]System.Delegate::Remove(**class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)**

 castclass CarDelegate.Car/AboutToBlow

 stfld class CarDelegate.Car/AboutToBlow CarDelegate.Car::almostDeadList

 ret

}

Если вызывающая сторона пожелает удалить элемент из списка вызовов делегата, мы должны предоставить тот же объект делегата, который был ранее добавлен. Поэтому мы можем прекратить выдачу сообщения Exploded, обновив Main() так, как показано ниже.

static void Main(string[] args) {

 Car cl = new Car("SlugBug", 100, 10);

**// Сохранение объекта Car.Exploded делегата.**

 Car.Exploded d = new Car.Exploded(CarExploded)*;*

 cl.OnExploded(d);

 …

**// Удаление метода CarExploded из списка вызовов.**

 cl.RemoveExploded(d);

 …

}

Вывод нашего приложения CarDelegate показан на рис. 8.5.

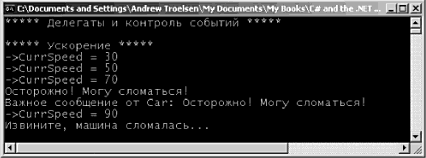


Рис. 8.5. Приложение CarDelegate за работой

Исходный код. Проект CarDelegate размещен в подкаталоге, соответствующем главе 8.

**Более совершенный пример делегата**

Чтобы продемонстрировать более интересные варианты использования делегатов, мы добавим в класс Car пару членов-переменных логического типа. Первый из добавляемых членов предназначен для индикации того, что автомобиль необходимо помыть (isDirty), а второй будет сообщать о том, что автомобиль требует замены шин (shouldRotate). Чтобы позволить пользователю объекта взаимодействовать с этими новыми данными состояния, для Car определяются необходимые дополнительные свойства и обновляется конструктор. Вот как выглядит соответствующим образом модифицированный программный код.

**// Обновленный класс Car.**

public class Car {

 …

**// Не пора ли помыть? Не пора ли сменить шины?**

 private bool isDirtу;

 private bool shouldRotate;

**// Дополнительные параметры для установки булевых значений.**

 public Car(string name, int maxSp, int currSp, **bool washCar, bool rotateTires**) {

  …

  isDirty = washCar;

  shouldRotate = rotateTires;

 }

 public bool Dirty {

  get { return isDirty; }

  set { isDirty = value; }

 }

 public bool Rotate {

  get { return shouldRotate; }

  set { shouldRotate = value; }

 }

}

Также предположим, что в тип Car вложен новый делегат CarDelegate.

**// Для Car определяется еще один делегат.**

public class Car {

 …

**// Может вызывать любой метод, получающий Car в виде параметра**

**// и не возвращающий ничего.**

 public delegate void CarDelegate(Car с);

 …

}

Здесь создается делегат с именем CarDelegate. Тип CarDelegate представляет "некоторую функцию", принимающую Car в качестве параметра и возвращающую пустое значение.

**Делегаты в качестве параметров**

Теперь, когда у нас есть новый тип делегата, который указывает на методы, получающие Car в виде параметра и не возвращающие ничего, мы можем создавать функции, которые принимают этот делегат в виде параметра. Для примера предположим, что у нас есть новый класс, которому назначено имя Garage (гараж). Этот тип поддерживает коллекцию типов Car, содержащихся в System.Collections. ArrayList. При создании ArrayList наполняется типами Car.

**// Класс Garage хранит список типов Car.**

using System.Collections;

 …

 public class Garage {

**// Создание списка всех машин в гараже.**

 ArrayList theCars = new ArrayList();

**// Создание машин в гараже.**

 public Garage() {

**// Напомним, что конструктор был обновлен,**

**// и теперь можно установить значения isDirty и shouldRotate**.

  theCars.Add(new Car("Viper", 100, 0, true, false));

  theCars.Add(new Car("Fred", 100, 0, false, false));

  theCars.Add(new Car("BillyBob", 100, 0, false, true));

 }

}

Класс Garage будет определять общедоступный метод ProcessCars(), который в качестве единственного аргумента получит новый тип делегата (Car.CarDelegate). В ProcessCars() каждый объект Car из коллекции будет передаваться в виде параметра "той функции, на которую указывает" делегат. При этом ProcessCars() использует члены Target и Method из System.MulticastDelegate, чтобы определить, на какую из функций делегат указывает в настоящий момент.

**// Класс Garage имеет метод, использующий CarDelegate.**

using System.Collections;

…

public class Garage {

 …

**// Этот метод получает Car.CarDelegate в виде параметра.**

 public void ProcessCars(Car.CarDelegate proc) {

**// Куда направить вызов?**

  Console.WriteLine("\*\*\*\*\* Вызывается: {0} \*\*\*\*\*", proc.**Method)**;

**// Вызывается метод экземпляра или статический метод?**

  if (proc.**Target**!= null) Console.WriteLine("-›Цель: {0} ", proc.**Target)**;

  else Console.WriteLine("-›Целевым является статический метод");

**// Вызов "указанного" метода всех машин по очереди.**

  foreach (Car с in theCars) {

**Console.**WriteLine("\n-› Обработка Car");

   proc(c);

  }

 }

}

Как и в случае любого делегата, при вызове ProcessCars() мы должны указать имя метода, который обработает запрос. Напомним, что такой метод может быть или статическим, или методом экземпляра. Для примера предположим, что в качестве такого метода будут использоваться члены экземпляра нового класса ServiceDepartment (отдел технического обслуживании), которым назначены имела WashCar() и RotateTires(). Обратите внимание на то, что эти два метода используют новые свойства Rotate и Dirty типа Car.

**// Этот класс определяет методы, которые будут вызываться**

**// типом Car.CarDelegate.**

public class ServiceDepartment {

 public void WashCar(Car c) {

  if (c.Dirty) Console.WriteLine("Моем машину");

  else Console.WriteLine("Эта машина уже помыта…");

 }

 public void RotateTires(Car с) {

  if (c.Rotate) Console.WriteLine("Меняем шины");

  else Console.WriteLine("Менять шины не требуется…");

 }

}

Теперь проиллюстрируем взаимодействие между новыми типами Car, CarDelegate, Garage и ServiceDepartment, рассмотрев их использование в следующем фрагменте программного кода.

**// Garage направляет все заказы в ServiceDepartment**

**// (найти хорошего механика всегда проблема…)**

public class Program {

 static void Main(string[] args) {

**// Создание гаража.**

  Garage g = new Garage();

**// Создание отделения обслуживания,**

  ServiceDepartment sd = new ServiceDepartment();

**// Garage моет машины и меняет шины,**

**// делегируя соответствующие полномочия ServiceDepartment.**

  g.ProcessCars(new Car.CarDelegate(sd.WashCar));

  g.ProcessCars(new Car.CarDelegate(sd.RotateTires));

  Console.ReadLine();

 }

}

На рис. 8.6 показан соответствующий вывод.

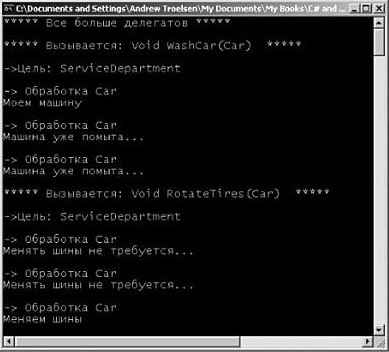


Рис. 8.6. Перекладывание ответственности

**Анализ программного кода делегирования**

Предложенный выше метод Main() начинается с создания экземпляров типов Garage и ServiceDepartment. Когда вы пишете

**// Помыть все грязные машины.**

g.ProcessCars(new Car.CarDelegate(sd.WashCar));

это на самом деле означает: "Добавить указатель на метод ServiceDepartment.WashCar() к объекту Car.CarDelegate и передать этот объект в Garage.ProcessCars()". Подобно любому автомобильному предприятию в реальном мире, все заказы передаются в отдел технического обслуживания (что и объясняет, почему замена масла, обычно требующая 30 минут, занимает целых 2 часа). С учетом этого ProcessCars() можно интерпретировать так.

**// CarDelegate указывает на функцию ServiceDepartment.WashCar**.

public void ProсessCars(Car.CarDelegate proc) {

 foreach (Car с in theCars)

 proc(c);**// proc(c) =› ServiceDepartment.WashCar(c)**

 …

}

Точно так же, если вы говорите

**// Поменять шины.**

g.ProcessCars(new Car.CarDelegate(sd.RotateTires));

то ProcessCars() можно интерпретировать, как

**// CarDelegate указывает на функцию ServiceDepartment.RotateTires.**

public void ProcessCars(Car.CarDelegate proc) {

 …

 foreach(Car с in the Cars)

 proc(c); **//proc(c) =› ServiceDepartment.RotateTires(e)**

 …

}

**Исходный код**. Проект CarGarage размещен в подкаталоге, соответствующем главе 8.

**Ковариантность делегатов**

К этому моменту вы должны чувствовать себя более уверенно при создании и использовании типов делегата. Перед тем как перейти к изучению синтаксиса событий в C#, мы рассмотрим новую возможность .NET 2.0, связанную с делегатами и обозначенную термином ковариантность. Вы могли обратить внимание на то, что все делегаты, созданные нами до сих пор, указывали на методы, возвращающие простые числовые типы данных (или не возвращающие значений вообще). Но предположим, что нам нужен делегат, способный указывать на методы, возвращающие пользовательский тип класса.

**// Определение делегата, который позволит указывать на объекты,**

**// возвращающие типы Car.**

public delegate Car ObtainCarDelegate();

Мы можем определить целевой объект для делегата так, как обычно.

class Program {

 public delegate Car ObtainCarDelegate();

 public static Car GetBasicCar() {return new Car();}

 static void Main(string[] args) {

  ObtainCarDelegate targetA = new ObtainCarDelegate(GetBasicCar);

  Car c = targetA();

  Console.ReadLine();

 }

}

Пока что все выглядит прекрасно. Но что делать, если мы получим новый класс SportsCar из типа Car и потребуется делегат, который сможет указывать на методы, возвращаемые этим новым типом класса? До появления .NET 2.0 в таком случае вам пришлось бы определить новый делегат.

**// Новый делегат, указывающий на целевые объекты,**

**// возвращающие типы SportsCar.**

public delegate SportsCar ObtainSportsCarDelegate();

У нас теперь два типа делегата, и мы должны создать по экземпляру каждого из них, чтобы получить типы Car и SportsCar.

class Program {

 public delegate **Car**ObtainCarDelegate();

 public delegate **SportsCar**ObtainSportsCarDelegate();

 public static **Car**GetBasicCar() {return new **Car();**}

 public static **SportsCar**GetSportsCar() {return new **SportsCar**();}

 static void Main(string[] args) {

  ObtainCarDelegate targetA = new ObtainCarDelegate(GetBasicCar);

**Car**с = targetA();

  ObtainSportsCarDelegate targetB = new ObtainSportsCarDelegate(GetSportsCar);

**SportsCar**sc = targetB();

  Console.ReadLine();

 }

}

По законам классического наследования в идеале лучше иметь один тип делегата, который мог бы указывать на методы, возвращающие либо тип Car, либо тип SportsCar (в конце концов, тип SportsCar связан с Car отношением наследования). *Ковариантность*позволяет реализовать именно такую возможность, т.е. построить один делегат, способный указывать на методы, возвращающие типы класса, связанные классическим отношением наследования.

class Program {

**// Определение делегата, способного возвращать**

**// как Car, так и SportsCar.**

 public delegate Car ObtainVehicalDelegate();

 public static Car GetBasicCar() {return new Car();}

 public static SportsCar GetSportsCar() { return new SportsCar();}

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Ковариантность делегатов \*\*\*\*\*\n");

  ObtainVehicalDelegate targetA = new ObtainVehicalDelegate(GetBasicCar);

  Car c = targetA();

**// Такое присваивание возможно вследствие ковариантности.**

  ObtainVehicalDelegate targetB = new ObtainVehicalDelegate(GetSportsCar);

  SportsCar sc = (SportsCar)targetB();

  Console.ReadLine();

 }

}

Обратите внимание на то, что тип делегата ObtainVehicalDelegate был определен для того, чтобы указывать на методы, возвращающие строго типизованный Car. Однако в условиях ковариантности мы получаем возможность указывать и на методы, возвращающие производные типы. Чтобы получить производный тип, нужно просто выполнить явное преобразование.

**Замечание.**Точно так же *ковариантность*обеспечивает возможность создания делегата, который позволит указать на множество методов, получающих объекты, связанные классическим отношением наследования. Более подробная информация имеется в документации .NET Framework 2.0 SDK.

**Исходный код.**Проект DelegateCovariance размещен в подкаталоге, соответствующем главе 8.

**События в C#**

Делегаты оказываются очень интересными конструкциями с той точки зрения, что они предоставляют возможность реализовать двухстороннее взаимодействие между объектами в памяти. Однако, и вы с этим согласитесь, работа с делегата-ми напрямую предполагает ввод больших по объему шаблонных фрагментов программного кода (определение делегата, объявление членов-переменных, создание пользовательских методов регистрации и отмены регистрации).

Поскольку возможность обратного вызови объектов другим объектом является очень полезной, в C# предлагается специальное ключевое слово event, позволяющее минимизировать неудобства программиста, связанные с непосредственным применением делегатов. При обработке ключевого слова event компилятор автоматически создает для вас методы регистрации и отмены регистрации, а также члены-переменные, необходимые для вашего типа делегата. Ключевое слово event Можно назвать синтаксической "конфеткой", позволяющей экономить время при вводе программного кода.

**Замечание.**Даже при использовании в C# ключевого слова event вам все равно придется вручную определять связанные с делегатом типы.

Процесс определения события состоит из двух шагов. Во-первых, вы должны определить делегат, который будет содержать методы, вызываемые при наступлении соответствующего события. Затем вы объявляете события (используя ключевое слово C# event) в терминах соответствующего делегата. Определение типа, способного посылать события, имеет следующий шаблон (записанный здесь в псевдокоде).

public class SenderOfEvents {

**public**delegate ***возврЗначение***AssociatedDelegate***(аргументы);***

**public event**AssociatedDelegate **ИмяСобытия;**

 …

}

События типа Car будут иметь те же имена, что и предыдущие делегаты (AboutToBlow и Exploded). Новому делегату, с которым будут ассоциироваться события, будет назначено имя CarEventHandler. Вот начальные изменения, вносимые в определение типа Car.

public class Car {

**// Этот делегат работает в связке с событиями Car**

 public delegate void CarEventHandler(string msg);

**// Объект Car может посылать эти события.**

 public event CarEventHandler Exploded;

 public event CarEventHandler AboutToBlow;

 …

}

Отправка событий вызывающей стороне выполняется с помощью простого указания имени события и всех обязательных параметров, предусмотренных в определении соответствующего делегата. Вы должны проверить событие на значение null перед тем, как вызывать набор методов делегата, чтобы гарантировать регистрацию события вызывающей стороной. С учетом этого предлагается новый вариант метода Accelerate() для типа Car.

public void Accelerate(int delta) {

**// Если машина сломана, генерируется событие Exploded.**

 if (carIsDead) {

  if (Exploded!= null) Exploded("Извините, машина сломалась…");

 } else {

  currSpeed += delta;

**// Вот-вот сломается?**

  if (10 == maxSpeed – currSpeed && AboutToBlow != null) {

   AboutToBlow ("Осторожно! Могу сломаться!");

  }

**// Пока все OK!**

  if (currSpeed ›= maxSpeed) carIsDead = true;

  else Console.WriteLine("-›CurrSpeed = {0}", currSpeed);

 }

}

Мы наделили автомобиль способностью посылать два типа пользовательских событии, избавив себя от необходимости определять пользовательские функции регистрации. Немного позже мы приведем пример использования нашего нового автомобили, но сначала более подробно рассмотрим архитектуру событий.

**Глубинный механизм событий**

Событие в C# представляется двумя скрытыми общедоступными методами, один из которых имеет префикс add\_, а другой – префикс remove\_. За этими префиксами следует имя события. Например, событие Exploded транслируется в пару CIL-методов с именами add\_Exploded() и remove\_Exploded(). Кроме приведения к методам add\_XXX() и remove\_XXX(), определение события на уровне CIL связывает данное событие с соответствующим делегатом.

Взгляните на CIL-инструкции для add\_AboutToBlow(), и вы обнаружите программный код, почти идентичный программному коду вспомогательного метода OnAboutToBlow() из рассмотренного выше примера CarDelegate (обратите внимание на строку с вызовом Delegate.Combine()).

.method public hidebysig specialname instance void **add\_AboutToBlow**(class CarEvents.Car/CarEventHandler 'value') cil managed synchronized {

 .maxstack 8

 ldarg.0

 ldarg.0

 ldfld class CarEvents.Car/CarEventHandler CarEvents.Car::AboutToBlow

 ldarg.1

 call class [mscorlib]System.Delegate [mscorlib] **System.Delegate::Combine**(class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)

 castclass CarEvents.Car/CarEventHandler

 stfld class CarEvents.Car/CarEventHandler

 CarEvents.Car::AboutToBlow

 ret

}

В соответствии с ожиданиями, метод remove\_AboutToBlow() неявно (опосредованно) вызывает Delegate.Remove() и приблизительно соответствует определенному выше вспомогательному методу RemoveAboutToBlow().

.method public hidebysig specialname instance void **remove\_AboutToBlow**(class CarEvents.Car/CarEventHandler 'value') cil managed synchronized {

 .maxstack 8

 ldarg.0

 ldarg.0

 ldfld class CarEvents.Car/CarEventHandler CarEvents.Car::AboutToBlow

 ldarg.1

 call class [mscorlib]System.Delegate [mscorlib]System.Delegate::Remove(class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)

 castclass CarEvents.Car/CarEventHandler

 stfld class CarEvents.Car/CarEventHandler CarEvents.Car::AboutToBlow

 ret

}

Наконец, программный код CIL, представляющий само событие, использует директивы .addon и .removeon для отображения имен в соответствующие имена вызываемых методов add\_XXX() и remove\_XXX().

.event CarEvents.Car/EngineHandler AboutToBlow {

**.addon**void CarEvents.Car::add\_AboutToBlow(class CarEvents.Car/CarEngineHandler)

**.removeon**void CarEvents.Car::remove\_AboutToBlow(class CarEvents.Car/CarEngineHandler)

}

Теперь, когда вы знаете, как строить классы, способные посылать события в C# (и знаете о том, что соответствующая событиям синтаксическая конструкция – это просто сокращение, позволяющее уменьшить объем вводимых с клавиатуры данных), мы должны выяснить, как осуществляется "прием" поступающих событий с точки зрения вызывающей стороны.

**Прием поступающих событий**

Использование событий в C# позволяет также упростить регистрацию обработчиков событий вызывающей стороны. Вместо необходимости указывать пользовательские вспомогательные методы, вызывающая сторона просто использует операции += и -= (которые в фоновом режиме "подключают" add\_XXX() или remove\_XXX()). Если вы хотите регистрировать событие, то следуйте показанному ниже шаблону.

**// ОбъектнаяПеременная.ИмяСобытия +=**

**// new СоответствующийДелегат(вызываемаяФункция);**

Car.EngineHandler d = new Car.EngineHandler(CarExplodedEventHandler) myCar.Exploded += d;

Чтобы отменить привязку к источнику событий, используйте операцию -=.

**// ОбъектнаяПеременная.ИмяСобытия -= объектДелегата**;

myCar.Exploded -= d;

С учетом этих соответствующих ожиданиям шаблонов, вот как должен выглядеть модифицированный метод Main(), в котором используется синтаксис регистрации событий C#.

class Program {

 statiс vоid Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* События \*\*\*\*\*");

  Car c1 = new Car("SlugBug", 100, 10);

**// Регистрация обработчиков событий.**

  сl.AboutToBlow += new Car.CarEventHandler(CarIsAlmostDoomed);

  cl.AboutToBlow += new Car.CarEventHandler(CarAbautToBlow);

  Car.CarEventHandler d = new Car.CarEventHandler(CarExploded);

  cl.Exploded += d;

  Console.WriteLine("\n\*\*\*\*\* Ускорение \*\*\*\*\*);

  for(int i = 0; i ‹ 6; i++) cl.Accelerate(20);

**// Удаление метода CarExploded из списка вызовов.**

  cl.Exploded -= d;

  Console.WriteLine("\n\*\*\*\*\* Ускорение \*\*\*\*\*");

  for(int i = 0; i ‹ 6; i++) cl.Accelerate(20);

  Console.ReadLine();

 }

 public static void CarAboutToBlow(string msg) { Console.WriteLine(msg); }

 public static void CarIsAlmostDoomed(string msg) { Console.WriteLine("Critical Message from Car: {0}", msg); }

 public static void CarExploded(string msg) { Console.WriteLine(msg); }

}

**Исходный код**. Проект CarEvents размещен в подкаталоге, соответствующем главе 8.

**Упрощенная регистрация событий в Visual Studio 2005**

В Visual Studio .NET 2003 и Visual Studio 2005 предлагается помощь в процессе регистрации обработчиков событий. При вводе += в окне программного кода появляется окно IntelliSense, предлагающее назвать клавишу *‹*Tab›*,*чтобы автоматически ввести соответствующий экземпляр делегата (рис. 8.7).



Рис. 8.7. Выбор делегата IntelliSense

После нажатия клавиши ‹Tab› будет предложено ввести имя генерируемого обработчика события (или согласиться использовать имя, предлагаемое по умолчанию), как показано на рис. 8.8.

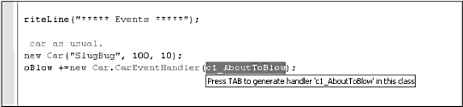


Рис. 8.8. Формат целевого объекта делегата IntelliSense

Если снова нажать клавишу ‹Tab›, вы получите программный код "заглушки" в нужном формате целевого объекта делегата (обратите внимание на то, что соответствующий метод объявляется статическим, поскольку событие было зарегистрировано с помощью статического метода).

static void cl\_AboutToBlow(string msg) {

 // Add your code!

}

Эта возможность IntelliSense доступна для всех событий .NET из библиотек базовых классов. Данная особенность интегрированной среды разработки позволяет разработчику существенно экономить время, поскольку избавляет от необходимости искать по справке .NET подходящие делегаты (и выяснять их форматы) для использования их с конкретными событиями.

**"Разборчивые" события**

Есть еще одно усовершенствование, которое можно внести в наш пример с CarEvents и которое соответствует шаблону событий, рекомендуемому разработчиками из Microsoft. При исследовании событий, посылаемых данным типом из библиотек базовых классов, вы обнаружите, что первым параметром соответствующего делегата является System.Object, а вторым – тип, производный от System.EventArgs.

Аргумент System.Object представляет ссылку на объект, посылающий событие (такой как, например, Car), а второй параметр представляет информацию о соответствующем событии. Базовый класс System.EventArgs представляет событие и не передает никакой пользовательской информации.

public class EventArgs {

 public static readonly System.EventArgs Empty;

 public EventArgs();

}

Для простых событий вы можете просто передать экземпляр EventArgs. Но если вы хотите передать и пользовательские данные, вы должны построить подходящий класс, производный от EventArgs. Для нашего примера мы предположим, что у нас есть класс CarEventArgs, который содержит строку с сообщением, отправляемым получателю.

public class CarEventArgs: **EventArgs**{

 public readonly string msg;

 public CarEventArgs(string message) {

  msg = message;

 }

}

Теперь мы должны обновить делегат CarEventHandler так, как показано ниже (события должны остаться без изменений).

public class Car {

 public delegate void CarEventHandler**(object sender, CarEventArgs e)**;

 …

}

При генерировании событий из метода Accelerate() мы теперь должны предоставить ссылку на текущий объект Car (с помощью ключевого слова **Car)**и экземпляр нашего типа CarEventArgs.

public void Accelerate(int delta) {

**// Если машина сломалась, генерируется событие Exploded.**

 if (carIsDead) {

  if (Exploded != null) Exploded(this, new CarEventArgs("Извините, машина сломалась…"));

  else {

   …

   AboutToBlow(this, new *C*arEventArgs("Осторожно! Могу сломаться!"));

  }

  …

 }

}

С точки зрения вызывающей стороны, все, что нам требуется, так это обновление обработчиков событий, чтобы иметь возможность принять поступающие параметры и получить сообщение через доступное только для чтения поле. Например:

public static void CarAboutToBlow(object sender, CarEventArgs e) { Console.WriteLine ("{0} сообщает: {1}", sender, e.msg); }

Если получатель желает взаимодействовать с объектом, отправившим событие, следует выполнить явное преобразование System.Object. Так, если нужно выключить радио, когда объект Car уже на полпути к своему создателю, можно предложить обработчик событий, который будет выглядеть примерно так.

public static void CarIsAlmostDoomed(object sender, CarEventArgs e) {

**// Просто для гарантии здесь перед вызовом предлагается**

**// проверка среды выполнения.**

 if (sender is Car) {

  Car с = (Car)sender;

  c.CrankTunes(false);

 }

 Console.WriteLine("Важное сообщение от {0}: {1}", sender, e.msg);

}

**Исходный код.**Проект PrimAndProperCarEvenfs размещен в подкаталоге, соответствующем главе 8.

**Анонимные методы в C#**

В завершение этой главы мы рассмотрим некоторые связанные с делегатами и событиями возможности .NET 2.0 через призму возможностей C#. Для начала обратим внимание на то, что в случае, когда вызывающей стороне требуется осуществлять прием поступающих событий, необходимо определить уникальный метод, отвечающий виду соответствующего делегата.

class SomeCaller {

 static void Main(string[] args) {

  SomeType t = new SomeType();

  t.SomeEvent += new SomeDelegate(MyEventHandler);

 }

**// Как правило, вызывается только объектом SomeDelegate.**

 public static void MyEventHandler() {…}

}

Если немного подумать, то станет ясно, что такие методы, как MyEventHandler(), редко бывают предназначены для вызова вне вызываемого делегата. А с точки зрения продуктивности слишком непривлекательно (хотя и не запрещено) вручную определять специальные методы, которые вызываются объектом делегата.

Чтобы разрешить эту проблему, теперь позволяется ассоциировать делегат непосредственно с блоком операторов программного кода при регистрации события. Формально такой программный код называется *анонимным методом.*В качестве иллюстрации базового синтаксиса рассмотрите следующий метод Main(), в кото-рам посылаемые типом Car события обрабатываются о помощью анонимных методов, а не специальных именованных программ обработки событий.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Анонимные методы \*\*\*\*\*\n");

  Car c1 = new Car("SlugBug", 100, 10);

**// Регистрация обработчиков событий с помощью**

**// анонимных методов.**

  c1.AboutToBlow += delegate {

   Console.WriteLine("Оx! Едем слишком быстро!");

  };

  c1.AboutToBlow += delegate(object sender, CarEventArgs e) {

   Console.WriteLine("Сообщение от Car: {0}", e.msg);

  };

  c1.Exploded += delegate(object sender, CarEventArgs e) {

   Console.WriteLine("Фатальное сообщение от Car: {0}", e.msg);

  };

  …

 }

}

Замечание. После завершающей фигурной скобки анонимного метода должна следовать точка с запятой. Если пропустить точку с запятой, будет получено сообщение об ошибке компиляции.

Обратите вниманий на то, что тип Program уже не определяет конкретные статические программы обработки событий, такие как, например, CarAboutToBlow() и CarExploded(). Вместо этого здесь указаны безымянные (т.е. анонимные) методы, определяемые "внутристрочно" в тот момент, когда вызывающая сторона обрабатывает событие, используя синтаксис +=.

Базовый синтаксис анонимного метода соответствует следующему представлению в псевдокоде.

class SomeCaller {

 static void Main(string[] args) {

  SomeType t = new SomeType();

  t.SomeEvent += delegate*(****необязательныеАргументыДелегата****)*{*/\**операторы \*/};

 }

}

В предыдущем варианте метода Main() следует обратить внимание на то, что при обработке первого события AboutToBlow мы не определяем аргументы, передаваемые делегатом.

c1.AboutToBlow += delegate {

 Console.WriteLine("Ox! Едем слишком быстро!");

};

Строго говоря, получать входные аргументы, посылаемые конкретным событием, не обязательно. Но если вы хотите использовать поступающие аргументы, вы должны указать параметры, прототипы которых заданы типом делегата (как это сделано во втором случае обработки событий AboutToBlow и Exploded). Например:

c1.AboutToBlow += delegate**(object sender, CarEventArgs e)**{

 Console.WriteLine("Важное сообщение от Car: {0}", e.msg);

};

**Доступ к "внешним" переменным**

Анонимные методы интересны в том отношении, что они позволяют доступ к локальным переменным определяющего их метода. Формально говоря, такие переменные являются "внешними переменными" анонимного метода. Для примера предположим, что наш метод Main() определяет локальную целую переменную-счетчик с именем aboutToBlowCounter. В рамках анонимных методов, обрабатывающих событие AboutToBlow, мы будем увеличивать этот счетчик на 1 и печатать его значение в конце Main().

static void Main(string[] args) {

 …

 int aboutToBlowCounter = 0;

**// Создание машины.**

 Car c1 = new Car("SlugBug", 100, 10);

**// Регистрация обработчиков событий в виде анонимных методов.**

 c1.AboutToBlow += delegate {

**aboutToBlowCounter++**;

  Console.WriteLine("Ox! Едем слишком быстро!");

 };

 c1.AboutToBlow += delegate(string msg) {

**aboutToBlowCounter++**;

  Console.WriteLine("Важное сообщение от Car: {0}", msg);

 };

 …

 Console.WriteLine("Событие AboutToBlow вызывалось {0} раз(а).", aboutToBlowCounter);

 Console.ReadLine();

}

В результате выполнения этого обновленного метода Main() завершающий оператор Console.WriteLine() сообщит вам о том, что событие AboutToBlow генерировалось дважды.

**Замечание.**Анонимный метод не имеет возможности получить доступ к параметрам ref и out определяющего метода.

**Групповое преобразование методов в C#**

Еще одной связанной с делегатами и событиями возможностью в C# является так называемое *групповое преобразование методов.*Эта возможность позволяет регистрировать "просто" имя обработчика событий. Чтобы пояснить это на примере, мы снова рассмотрим тип SimpleMath, уже рассматривавшийся в этой главе выше, но добавим в него новое событие, которому будет назначено имя ComputationFinished.

public class SimpleMath {

**// Здесь мы не утруждаем себя созданием**

**// производного типа System.EventArgs.**

 public delegate void MathMessage(string msg);

 public event MathMessage ComputationFinished;

 public int Add(int x, int y) {

  ComputationFinished("Сложение выполнено.");

  return х + y;

 }

 public int Subtract(int x*,*int y) {

  ComputationFinished("Вычитание выполнено.");

  return x – у;

 }

}

Если не использовать синтаксис анонимных методов, то мы должны обработать событие ComputationComplete так, как показано ниже.

class Program {

 static void Main(string[] args) {

  SimpleMath m = new SimpleMath();

  m.ComputationFinished += new SimpleMath.MathMessage(ComputationFinishedHandler);

  Console.WriteLine("10 + 10 равно {0}", m.Add(10, 10));

  Console.ReadLine();

 }

 static void ComputationFinishedHandler(string msg) { Console.WriteLine(msg); }

}

Но можно зарегистрировать программу обработки событий для конкретного события и так, как показано ниже (в остальном программный код изменений не претерпевает).

m.ComputationFinished += **ComputationFinishedHandler**;

Обратите внимание на то, что мы не создаем непосредственно соответствующий тип делегата, а просто указываем метод, который соответствует ожидаемой сигнатуре делегата (в данном случае это метод, не возвращающий ничего и получающий один объект типа System.String). Ясно, что компилятор C# при этом должен обеспечить типовую безопасность. Если метод ComputationFinishedHandler() не получает System.String и не возвращает void, то вы получите сообщение об ошибке компиляции.

Можно и явно конвертировать обработчик события в экземпляр соответствующего делегата. Это может оказаться полезным тогда, когда нужно получить соответствующий делегат, использующий заранее определенный метод. Например:

**// .NET 2.0 допускает преобразование обработчиков событий**

**// в соответствующие делегаты.**

SimpleMath.MathMessage mmDelegate = (SimpleMath.MathMessage)ComputationFinishedHandler;

Console.WriteLine(mmDelegate.Method);

Если выполнить этот программный код, то заключительный оператор Console.WriteLine() напечатает сигнатуру ComputationFinishedHandler, как показано на рис. 8.9.



Рис. 8.9. Можно извлечь делегат из соответствующего обработчика события

**Исходный код**. Проект AnonymousMethods размещен в подкаталоге, соответствующем главе 8.

**Резюме**

В этой главе был рассмотрен ряд подходов, позволяющих реализовать возможность двухстороннего взаимодействия объектов. Сначала было рассмотрено использование *интерфейсов обратного вызова,*которые обеспечивают возможность для объекта А вызывать объект В с помощью общего интерфейса. Этот подход не является специфическим для .NET, а может использоваться в любых языках и на любых платформах, допускающих программирование на основе интерфейсов.

Затем было рассмотрено ключевое слово C# delegate, которое используется для непрямого построения классов, производных от System.MulticastDelegate. Как выяснилось, делегат представляет собой объект, хранящий список методов, доступных для вызова. При этом вызовы, могут быть синхронными (они выполняются с помощью метода Invoke()) или асинхронными (они выполняются с помощью методов BeginInvoke() и EndInvoke()). Асинхронная природа типов делегата .NET будет рассмотрена позже.

Ключевое слово C# event при использовании с типом делегата позволяет упростить процесс отправки сообщений событий вызывающим объектам. Как показывает генерируемый CIL-код, модель событий .NET сводит ситуацию к скрытым вызовам типов System.Delegate/System.MulticastDelegate. В этой связи ключевое слово C# event оказывается необязательным и просто экономит время при наборе текста программы.

Новая возможность, появившаяся в C# 2005 и получившая название *анонимных методов*, позволяет непосредственно ассоциировать с событием (неименованный) блок операторов программного кода. Анонимные методы могут игнорировать параметры, посылаемые событием, и получать доступ в "внешним переменным" определяющего метода. В завершение главы был рассмотрен упрощенный способ регистрации событий с помощью *группового преобразования методов.*

**ГЛАВА 9. Специальные приемы построения типов**

В этой главе вы расширите горизонты вашего понимания языка C#, рассмотрев ряд более сложных (но весьма полезных) синтаксических конструкций. Сначала мы с вами выясним, как использовать метод индексатора. Этот механизм в C# позволяет строить пользовательские типы, обеспечивающие доступ к внутренним подтипам на основе синтаксиса массивов. Научившись строить методы индексатора, вы затем узнаете, как перегружать различные операции (+, -, ‹, › и т.д.) и явно или неявно создавать пользовательские подпрограммы преобразования типов (а также узнаете, зачем это может понадобиться).

Во второй половине главы будет рассмотрен небольшой набор ключевых слов C#, которые позволяют реализовать весьма интересные конструкции, хотя используются не очень часто. Вы узнаете о том, как с помощью ключевых слов checked и unchecked программно учитывать условия переполнения и потери значимости, а также о том, как создается "небезопасный" программный контекст, обеспечивающий возможность непосредственного управления ссылочными типами в C#. Завершается глава обсуждением роли директив препроцессора C#.

**Создание пользовательских индексаторов**

Как программисты, мы прекрасно знаем, что с помощью индексов можно получить доступ к отдельным элементам, содержащимся в стандартном массиве.

**// Объявление массива целых значений.**

int[] myInts = {10, 9, 100, 432, 9874};

**// Использование операции [] для доступа к элементам.**

for (int j = 0; j ‹ myInts.Length; j++) Console.WriteLine("Индекс {0} = {1}", j, myInts[j]);

Этот программный код ни в коем случае не претендует на новизну. Но язык C# дает возможность строить пользовательские классы и структуры, которые могут индексироваться подобно стандартным массивам. Поэтому совсем не удивительно, что метод, который обеспечивает такой доступ к элементам, называется *индекса-mopoм.*

Перед тем как приступить к созданию соответствующей конструкции, мы рассмотрим один пример. Предположим, что поддержка метода индексатора уже добавлена в пользовательскую коллекцию Garage (гараж), уже рассматривавшуюся в главе 8. Проанализируйте следующий пример ее использования.

**// Индексаторы обеспечивают доступ к элементам подобно массивам.**

public class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Забавы с индексаторами \*\*\*\*\*\n");

**// Предположим, что Garage имеет метод индексатора.**

  Garage carLot = new Garage();

**// Добавление в гараж машин с помощью индексатора.**

  сarLot[0] = new Саr("FееFee", 200);

  carLot[1] = new Car("Clunker", 90);

  carLot[2] = new Car("Zippy", 30);

**// Чтение и отображение элементов с помощью индексатора.**

  for (int i = 0; i ‹ 3; i++) {

   Console.WriteLine("Hомep машины: {0}", i);

   Console.WriteLite("Нaзвaниe: {0}", carLot[i].PetName);

   Console.WriteLine("Максимальная скорость: {0}", carLot[i].CurrSpeed);

   Console.WriteLine();

  }

  Console.ReadLine();

 }

}

Как видите, индексаторы ведут себя во многом подобно пользовательской коллекции, поддерживающей интерфейсы IEnumerator и IEnumerable. Основное различие в том, что вместо доступа к содержимому посредством типов интерфейса вы можете работать с внутренней коллекцией автомобилей, как с обычным массивом.

Здесь возникает вопрос: "Как сконфигурировать класс (или структуру), чтобы обеспечить поддержку соответствующих функциональных возможностей?" Индексатор в C# представляет собой несколько "искаженное" свойство. Для создания индексатора в самой простой форме используется синтаксис this[]. Вот как может выглядеть подходящая модификации типа Garage.

**// Добавление индексатора в определение класса.**

public class Garage: IEnumerable { **// для каждого элемента**

 …

**// Использование ArrayList для типов Car.**

 private ArrayList carArray = new ArrayList();

**// Индексатор возвращает тип Car, соответствующий**

**// Числовому индексу.**

 public Car this[int pos] {

**// ArrayList тоже имеет индексатор!**

  get { return (Car)carArray[pos]; }

  set { carArray.Add(value); }

 }

}

Если не обращать внимания на ключевое слово this, то объявление индексатора очень похоже на объявление свойства в C#. Но следует подчеркнуть, что индексаторы не обеспечивают иных функциональных возможностей массива, кроме возможности использования операции индексирования, Другими словами, пользователь объекта не может применить программный код, подобный следующему.

**// Используется свойство ArrayList.Count? Нет!**

Console.WriteLine("Машин в наличии: {0} ", carLot.**Count**);

Для поддержки этой функциональной возможности вы должны добавить свое свойство Count в тип Garage и, соответственно, делегат.

public class Garage: IEnumerable {

 …

**// Локализация/делегирование в действии снова.**

 public int Count { get { return carArray.Count; } }

}

Итак, индексаторы – это еще одна синтаксическая "конфетка", поскольку соответствующих функциональных возможностей можно достичь и с помощью "обычных" методов. Например, если бы тип Garage не поддерживал индексатор, все равно можно было бы позволить "внешнему миру" взаимодействовать с внутренним массивом, используя для этого именованное свойство или традиционные методы чтения и модификации данных (accessor/mutator). Но при использовании индексаторов пользовательские типы коллекции лучше согласуются со структурой библиотек базовых классов .NET.

**Исходный код.**Проект SimpleIndexer размещен в подкаталоге, соответствующем главе 9.

**Вариации индексатора для типа Garage**

В своем текущем виде тип Gаrage определяет индексатор, который позволяет вызывающей стороне идентифицировать внутренние элементы, используя число-вое значение. Но это не является непременным требованием метода индексатора. Предположим, что объекты Car содержатся в System.Collections.Specialized. ListDictionary, а не в ArrayList. Поскольку типы ListDictionary позволяют доступ к содержащимся типам с помощью ключевых маркеров (таких как, например, строки), можно создать новый индексатор Garage, подобный показанному ниже.

public class Garage: IEnumerable {

 private ListDictionary carDictionary = new ListDictionarу();

**// Этот индексатор возвращает соответствующий тип Car**

**// на основе строкового индекса.**

 public Car this[string name] {

  get { return (Car)carDictionary[name]; }

  set { carDictionary[name] = value; }

 }

 public int Length { get { return carDictionary.Count; } }

 public IEnumerator GetEnumerator() { return carDictionary.GetEnumerator(); }

}

Вызывающая сторона теперь может взаимодействовать с машинами внутри так, как показано ниже,

public class Program {

 static void Main(string[] args) {

  Console:WriteLine("\*\*\*\*\* Забавы с индексаторами \*\*\*\*\*\n");

  Garage carLot = new Garage();

**// Добавление именованных машин в гараж.**

  carLot["FeeFee"] = new Car("FeeFee", 200, 0);

  carLot["Clunker"] = new Car("Clunker", 90, 0);

  carLot["Zippy"] = new Car("Zippy", 30, 0);

**// Доступ к Zippy.**

  Car zippy = carLot["Zippy"];

  Console.WriteLine("{0} едет со скоростью {1} км/ч", zippy.PetName, zippy.CurrSpeed);

  Console.ReadLine();

 }

}

Индексаторы могут быть и перегруженными. Так, чтобы позволить вызывающей стороне доступ к внутренним элементам посредством числового индекса или строковых значений, вы можете определить множество индексаторов для одного типа.

**Исходный код.**Проект StringIndexer размещен в подкаталоге, соответствующем главе 9.

**Внутреннее представление индексаторов типов**

Мы рассмотрели примеры метода индексатора в C#, и пришло время выяснить, как представляются индексаторы в терминах CIL. Если открыть числовой индексатор типа Garage, то будет видно, что компилятор C# создает свойство Item, которое сводится к подходящей паре методов get/set.

property instance class SimpleIndexer.Car **Item**(int32) {

 .get instance class SimpleIndexer.Car SimpleIndexer.Garage::get\_Item(int32)

 .set instance void SimpleIndexer.Garage::set\_Item(int32, class SimpleIndexer.Car)

} // end of property Garage::Item

Методы get\_Item() и set\_Item() будут реализованы аналогично любому другому свойству .NET, например:

method public hidebysig specialname instance сlass SimpleIndexer.Car get\_Item(int32 pos) cil managed {

 Code size 22 (0x16)

 .maxstack 2

 .locals init ([0] class SimpleIndexer.Car CSS1$0000)

 IL\_0000: ldarg.0

 IL\_0001: ldfld class [mscorlib] System.Collections.ArrayList SimpleIndexer.Garage::carArray

 IL\_0006: ldarg.1

 IL\_0007: callvirt instance object [mscorlib]  Sysftem.Collections.ArrayList::get\_Item(int32)

 IL\_000c: castclass SimpleIndexer.Car

 IL\_0011: stloc.0

 IL\_0012: br.s IL\_0014

 IL\_0014: ldloc.0

 IL\_0015: ret

} // end of method Garage::get\_Item

**Заключительные замечания об индексаторах**

Чтобы получить настоящую экзотику, вы можете создать индексатор, который имеет множество параметров. Предположим, что у нас есть пользовательская кол-лекция, которая хранит элементы в двумерном массиве. В этом случае вы можете создать метод индексатора, доказанный ниже.

public class SameContainer {

 private int[,] my2DinArray = new int[10, 10];

 public int this[int row, int column] {/\* прочитать или установить значение 2D-массива \* /}

}

В заключение следует заметить, что индексаторы могут определяться и для типа интерфейса .NET, что обеспечивает реализующим интерфейс типам возможность его настройки. Вот пример такого интерфейса.

public interface IEstablishSubObjects {

**// Этот интерфейс определяет индексатор*,*возвращающий**

**// строки на основе числового индекса.**

 string this[int index] {get; set;}

}

Пожалуй, об индексаторах C# уже сказано достаточно. Перейдем к рассмотрению еще одного подхода, используемого в некоторых (но не во всех) языках программирования .NET: это перегрузка операций.

**Перегрузка операций**

В C#, как и в любом другом языке программирования, есть свой ограниченный набор лексем, используемых для выполнения базовых операций со встроенными типами. Так, вы знаете, что операция + применима к двум целым числам и в результате дает их сумму.

**// Операция + с целыми числами.**

int а = 100;

int b = 240;

int с = **а + b**; // с теперь равно 340

Снова заметим, что это не новость, но вы, наверное, заметили и то, что одна и та же операция + может применяться к большинству встроенных типов данных C#. Рассмотрите, например, следующий фрагмент программного кода.

**// Операция + со строками.**

string s1 = "Hello";

string s2 = " world!";

string s3 = **s1 + s2**; // s3 теперь равно "Hello world!"

По сути, операция + функционирует уникальным образом в зависимости от поставляемых типов данных (в данном случае это строки или целые числа). Когда операция + применяется к числовым типам, результатом является сумма операндов, а когда операция + применяется к строковым типам, результатом будет конкатенация строк.

Язык C# обеспечивает возможность построения пользовательских классов и структур, которые будут по-своему отвечать на один и тот же набор базовых лексем (таких, как операция +). При этом следует заметить, что можно "перегружать" не все встроенные операции C#. В табл. 9.1 указаны возможности перегрузки базовых операций.

**Таблица 9.1.**Возможности перегрузки операций

|  |  |
| --- | --- |
| **Операции C#** | **Возможность перегрузки** |
| +, -, !, ~, ++, --, true, false | Эти унарные операции допускают перегрузку |
| +, -, \*, /, %, &, |, ^, ‹‹, ›› | Эти бинарные операции допускают перегрузку |
| ==, !=, ‹, ›, ‹=, ›= | Операции сравнения допускают перегрузку. В C# требуется, чтобы перегрузка "родственных" операций (т.е. ‹ и *›,*‹= и ›=, == и !=) выполнялась одновременно |
| [] | Операция [] не допускает перегрузку. Но, как было показано выше, аналогичные перегрузке возможности обеспечивает конструкция индексатора |
| () | Операция () не допускает перегрузку. Но, как будет показано ниже, аналогичные перегрузке возможности обеспечивают пользовательские методы преобразования |
| +=, -=, \*=, /=, %=, &=, |=, ^=, ‹‹=, ››= | Операторные сокращения с присваиванием сами по себе не допускают перегрузку, однако для них перегруженная форма получается автоматически в результате перегрузки соответствующей бинарной операции |

**Перегрузка бинарных операций**

Чтобы проиллюстрировать процесс перегрузки бинарных операций, расcмо-трим следующую простую структуру Point (точка).

**// Самая обычная структура C#.**

public struct Point {

 private int x, y;

 public Point(int xPos, int yPos) {

  x = xPos*;*у = yPos;

 }

 public override string ToString() {

  return string.Format("[{0}, {1}]", this.x, this.у);

 }

}

Можно, скажем, рассмотреть сложение типов Point. Можно также вычесть один тип Point из другого. Например, вы можете записать, следующий программный код.

**// Сложение и вычитание двух точек.**

static vоid Main(string [] args) {

 Console.WriteLine("\*\*\* Забавы с перегруженными операциями \*\*\*\n");

**// Создание двух точек.**

 Point ptOne = new Point(100, 100);

 Point ptTwo = new Point (40, 40);

 Console.WriteLine("ptOne = {0}", ptOne);

 Console.WriteLine("ptTwo = {0}", ptTwo);

**// Сложение точек в одну большую точку?**

 Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

**// Вычитание одной точки из другой дает меньшую точку?**

 Console.WriteLine("ptOne – ptTwo: {0} ", ptOne – ptTwo);

 Console.ReadLine();

}

Чтобы позволить пользовательскому типу по-своему отвечать на встроенные операции, в C# предлагается ключевое слово operator, которое можно использовать только со *статическими* методами. Перегруженной бинарной операции (такой, как + или -) на вход подаются два аргумента, которые имеют тип определяющего класса (в данном примере это Point), как показывает следующий программный код.

// Более 'интеллектуальный' тип Point.

public struct Point {

 …

**// перегруженная операция +**

 public static Point operator+(Point p1, Point p2) { return new Point(p1.x + p2.x, p1.y + p2.y); }

**// перегруженная операция -**

 public static Point operator–(Point p1, Point p2) { return new Point(p1.x – p2.x, p1.y – p2.y); }

}

По логике операция + должна просто возвратить совершенно новый объект Point, полученный в результате суммирования соответствующих полей входных параметров Point. Поэтому, когда вы пишете pt1 + pt2, можете представлять себе следующий скрытый вызов статического метода операции +.

**// р3 = Point.операция+(p1, р2)**

р3 = p1 + р2;

Точно так же p2 – p2 отображается в следующее.

**// р3 = Point.операция-(p1, р2)**

р3 = p1 – p2;

**Операции += и -=**

Если вы изучаете C#, уже имея опыт использования C++, то можете обратить внимание на отсутствие возможности перегрузки операторных сокращений, включающих операцию присваивания (+=, -= и т.д.). Не волнуйтесь, в C# операторные сокращения с присваиванием моделируются автоматически, если тип предполагает перегрузку соответствующей бинарной операции. Поэтому, поскольку структура Point уже использует перегрузку операций + и -, вы можете записать следующее.

**// Перегрузка бинарных операций автоматически влечет перегрузку**

**// операторных сокращений с присваиванием.**

static void Main(string[] args) {

**// Автоматическая перегрузка +=**

 Point ptThree = new Point(90, 5);

 Console.WriteLine("ptThree = {0}", ptThree);

 Console.WriteLine("ptThree +=ptTwo: {0}*",*ptThree += ptTwo);

**// Автоматическая перегрузка -=**

 Point ptFour = new Point(0, 500);

 Console.WriteLine("ptFour = {0}", ptFour);

 Console.WriteLine("ptFour -= ptThree: {0}", ptFour -= ptThree);

}

**Перегрузка унарных операций**

В C# также позволяется перегрузка унарных операций, таких как, например, ++ и --. При перегрузке унарной операции вы тоже должны с помощью ключевого слова operator определить статический метод, но в данном случае передается только один параметр, который должен иметь тип, соответствующий определяющему классу или структуре. Например, если добавить в Point следующие перегруженные операции

public struct Point {

 …

**// Добавление 1 к поступившему Point.**

 public static Point operator++(Point p1) { return new Point(p1.x+1, p1.y+1); }

**// Вычитание 1 от поступившего Point.**

 public static Point operator--(Point p1) { return new Point(p1.x-1, p1.y-1); }

}

то вы получите возможность увеличивать или уменьшать на единицу значения X и Y объекта Point, как показано ниже.

static void Main(string[] args) {

 …

**// Применение унарных операций *++*и -- к Point.**

 Console.WriteLine("++ptFive = {0}", ++ptFive);

 Console.WriteLine("--ptFive = {0}", --ptFive);

}

**Перегрузка операций проверки на тождественность**

Вы можете помнить из материала главы 3, что System.Object.Equals() можно переопределить, чтобы сравнение типов выполнялось на основе значений (а не ссылок). Если вы переопределите Equals() (и связанный с Equals() метод System.Object.GetHashCode()), то будет очень просто задать перегрузку операций проверки на тождественность (== и !=). Для иллюстрации мы рассмотрим обновленный тип Point.

**// Такая 'инкарнация' Point задает также перегрузку операций == и !=*.***

public struct Point {

 …

 public override bool Equals(object o) {

  if (o is Point) {

   if (((Point)o).x == this.x && ((Point)о). у == this.y) return true;

  }

  return false;

 }

 public override int GetHashCode() { return this.ToString().GetHashCode(); }

**// Здесь позволяется перегрузка операций == и !=.**

 public static bool operator==(Point p1, Point p2) { return p1.Equals(p2); }

 public static bool operator!=(Point p1, Point p2) { return!p1.Equals(p2); }

}

Обратите внимание на то, что данная реализация операций == и != просто вызывает переопределенный метод Equals(), который и выполняет основную работу. С учетом этого вы можете теперь использовать свой класс Point так.

**// Использование перегруженных операций проверки на тождественность.**

static void Main(string[] args) {

 …

 Console.WriteLine("ptOne == ptTwo: {0}", **ptOne == ptTwo)**;

 Console.WriteLine("ptOne != ptTwo: {0}", **ptOne != ptTwo);**

}

Как видите, здесь два объекта сравниваются с помощью операций == и !=, а не с помощью "менее естественного" вызова Object.Equals(). При использовании перегрузки операций проверки на тождественность для класса имейте в виду, что в C# требуется, чтобы при переопределении операции – *обязательно*переопределялась и операция != (если вы забудете это сделать, компилятор вам напомнит).

**Перегрузка операций сравнения**

Из материала главы 7 вы узнали о том, как реализовать интерфейс IComparable, чтобы иметь возможность сравнения подобных объектов. В дополнение к этому для того же класса вы можете использовать перегрузку операций сравнения (‹, ›, ‹= и ›=). Подобно операциям проверки на тождественность, в C# требуется, чтобы при перегрузке ‹ выполнялась и перегрузка ›. Это же касается и операций ‹= и ›=. Если тип Point использует перегрузку операций сравнения, пользователь объекта получает возможность сравнивать объекты Point так, как показано ниже.

**// Использование перегруженных операций ‹ и ›.**

static void Main(string[] args) {

 …

 Console.WriteLine("ptOne ‹ ptTwo: {0}*",***ptOne ‹ ptTwo)**;

 Console.WriteLine("ptOne › ptTwo: {0}", **ptOne › ptTwo);**

}

В предположении о том, что интерфейс IComparable реализован, перегрузка операций сравнения оказывается тривиальной. Вот как может выглядеть обновленное определение класса.

**// Можно сравнивать объекты Point с помощью операций сравнения.**

public struct Point: **IComparable**{

 …

 public int CompareTo(object obj) {

  if (obj is Point) {

   Point p = (Point)obj;

   if (this.x › p.x && this.y › p.y) return 1;

   if (this.x ‹ p.x && this.y ‹ p.y) return -1;

   else return 0;

  } else throw new ArgumentException();

 }

 public static bool operator‹(Point p1, Point p2) { return(p1.CompareTo(р2) ‹ 0); }

 public static bool operator›(Point p1, Point p2) { return(p1.CompareTo(p2) › 0); }

 public static bool operator‹=(Point p1, Point p2) { return(p1.CompareTo(p2) ‹= 0); }

 public statiс bool operator›=(Point p1, Point p2) { return(p1.CompareTo(p2) ›= 0); }

}

**Внутреннее представление перегруженных операций**

Подобно любому элементу программы C#, перегруженные операции представляются специальными элементами синтаксиса CIL. Откройте, например, компоновочный блок OverloadedOps.exe с помощью ildasm.exe. Как показано на рис. 9.1, перегруженные операции внутри блока представляются скрытыми методами (это, например, op\_Addition(), oр\_Subtraction(), op\_Equality() и т.д.).

Теперь, если рассмотреть CIL-инструкции для метода op\_Addition, то вы обнаружите, что csc.exe добавляет в метод ключевое слово specialname.

.method public hidebysig **specialname**static valuetype OverloadedOps.Point op\_Addition(valuetype OverloadedsOps.Point p1, valuetype OverloadedOps.Point p2) cil managed {

 …

}

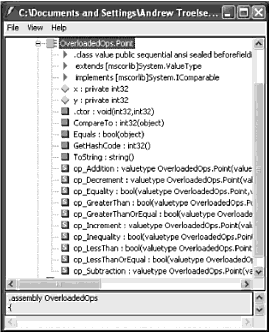


Рис. 9.1. В терминах CIL перегруженные операции отображаются в скрытые методы

Итак, любая операция, допускающая перегрузку, сводится в терминах CIL к специальному именованному методу. В табл. 9.2 раскрывается соответствие имен типичных операций C# и методов CIL.

**Таблица 9.2.**Соответствие имен операций C# и методов CIL

|  |  |
| --- | --- |
| **Внутренняя операция C#** | **Представление CIL** |
| **–-** | op\_Decrement() |
| ++ | op\_Increment() |
| + | op\_Addition() |
| – | op\_Subtraction() |
| \* | op\_Multiply() |
| / | op\_Division() |
| == | op\_Equality() |
| › | op\_GreaterThan() |
| ‹ | op\_LessThan() |
| != | op\_Inequality() |
| ›= | op\_GreaterThanOrEqual() |
| ‹= | op\_LessThanOrEqual() |
| –= | op\_SubtractionAssignment() |
| += | op\_AdditionAssignment() |

**Использование перегруженных операций в языках, не поддерживающих перегрузку операций**

Понимание того, как перегруженные операции представлены в программном коде CIL интересно не только с академической точки зрения. Чтобы осознать практическую пользу этих знаний, вспомните о том, что возможность перегрузки операций поддерживается не всеми языками, предназначенными для .NET. Как, например, добавить пару типов Point в программу, созданную на языке, не поддерживающем перегрузку операций?

Одним из подходов является создание "нормальных" открытых членов, которые будут решать ту же задачу, что и перегруженные операции. Например, можно добавить в Point методы Add() в Subtract(), которые будут выполнять работу, соответствующую операциям + и -.

**// Экспозиция семантики перегруженных операций**

**// с помощью простых членов-функций.**

public struct Point {

**// Представление операции + с помощью Add()**

 public static Point Add(Point p1, Point p2) { return p1 + p2; }

**// Представление операции – с помощью Subtract()**

 public static Point Subtract(Point p1, Paint p2) { return p1 – p2; }

}

С такими модификациями тип Point способен демонстрировать соответствующие функциональные возможности, используя любые подходы, предлагаемые в рамках данного языка. Пользователи C# могут применять операции + и – или же вызывать Add()/Subtract().

**// Использование операции + или Add()**,

Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

Console.WriteLine("Point.Add(ptOne, ptTwo): {0} ", Point.Add(ptOne, ptTwo));

**// Использование операции – или Subtract().**

Console.WriteLine("ptOne – ptTwo: {0} ", ptOne – ptTwo);

Console.WriteLine("Point.Subtract(ptOne, ptTwo): {0} ", Point.Subtract(ptOne, ptTwo));

Языки, в которых не допускается перегрузка операций, могут использовать только открытые статические методы. В качестве альтернативы можно предложить непосредственный вызов специальных именованных методов, создаваемых компилятором.

Рассмотрим исходный вариант языка программирования VB. NET. При построении консольного приложения VB .NET, ссыпающегося на тип Point, вы можете добавлять или вычитать типы Point, используя "специальные CIL-имена", например:

**' Предполагается, что данное приложение VB.NET ' имеет доступ к типу Point.**

Module OverLoadedOpClient

 Sub Main()

  Dim p1 As Point

  p1.x = 200

  p1.y= 9

  Dim p2 As Point

  p2.x = 9

  p2.y = 983

**' He так красиво, как вызов AddPoints(),**

'**но зато работает.**

  Dim bigPoint = **Point.op\_Addition**(p1**,**p2)

  Console.WriteLine("Большая точка {0}", bigPoint)

 End Sub

End Module

Как видите, языки программирования .NET, не предусматривающие перегрузку операций, способны непосредственно вызывать внутренние методы CIL, как "обычные" методы. Такое решение нельзя назвать слишком "изящным", но оно работает.

**Замечание.**Текущая версия VB .NET (Visual Basic .NET 2005) перегрузку операций поддерживает. Однако для других (многочисленных) управляемых языков, не поддерживающих перегрузку операций, знание "специальных имен" соответствующих методов CIL может оказаться очень полезным.

**Заключительные замечания о перегрузке операций**

Вы могли убедиться в том, что C# обеспечивает возможность построения типов, по-своему отвечающих на встроенные всем известные операции. Перед тем как перейти к непосредственной модификации классов для поддержки такого поведения, вы должны убедиться в том, что для операций, которым вы хотите назначить перегрузку, такая перегрузка имеет смысл.

Предположим, например, что вы хотите использовать перегрузку операции умножения для класса Engine (мотор). Что тогда должно означать умножение двух объектов Engine? He понятно. Перегрузка операций, в общем, оказывается полезной только тогда, когда строятся полезные типы. Строки, точки, прямоугольники и шестиугольники являются хорошими объектами для перегрузки операций. А люди, менеджеры, автомобили, наушники и бейсбольные кепки – нет. Если перегруженная операция делает более *трудным*понимание функциональных возможностей типа пользователем, то лучше перегрузку не использовать. Используйте указанную возможность с умом.

**Исходный код.**Проект OverloadedOps размещен в подкаталоге, соответствующем главе 9.

**Пользовательские преобразования типов**

Рассмотрим тему, близко связанную с перегрузкой операций: это пользовательские правила преобразования типов. В начале вашего рассмотрения мы кратко обсудим явные и неявные преобразования числовых данных и соответствующих типов класса.

**Преобразования чисел**

В случае встроенных числовых типов (sbyte, int, float и т.д.) *явное преобразование*требуется тогда, когда вы пытаетесь сохранить большее значение в меньшем контейнере, поскольку при этом может происходить потеря данных. По сути, это способ сказать компилятору примерно следующее: "Не беспокойся, я знаю, что делаю!" С другой стороны, неявное преобразование происходит автоматически, когда вы пытаетесь разместить в типе-адресате тип меньших размеров, в результате чего потери данных не происходит.

static void Main() {

 int a = 123;

 long b = a; **// Неявное преобразование из int a long**

 int с**=**(int)b**; // Явное преобразование из long в int**

}

**Преобразования типов класса**

Как показано в главе 4, типы класса могут быть связаны классическим отношением наследования (отношение "is-a"). В этом случае в C# процесс преобразования позволяет сдвигаться вверх или вниз по иерархии классов. Например, производный класс всегда можно неявно преобразовать в базовый тип. Однако если вы захотите сохранить базовый тип класса в производной переменной, придется выполнить явное преобразование.

**// Два связанных типа класса.**

class Base{}

class Derived: Base{}

class Program {

 static void Main() {

**// Неявное преобразование из производного в базовый.**

  Base myBaseType;

  myBaseType = new Derived();

**// Для сохранения базовой ссылки в производном типе**

**// следует выполнить явное преобразование**.

  Derived myDerivedType = (Derived)myBaseType;

 }

}

Здесь явное преобразование работает благодаря тому, что классы Base и Derived связаны классическим отношением наследования. Но что делать в том случае, когда вы хотите связать преобразованием два типа класса, принадлежащие *разным иерархиям?*Если классы не связаны классическим наследованием, явное преобразование помочь ничем не сможет.

В соответствующем ключе рассмотрим типы, характеризуемые значениями. Предположим, что у нас есть две .NET-структуры с именами Square (квадрат) и Rectangle (прямоугольник). Поскольку структуры не могут использовать классическое наследование, нет и естественного способа взаимного преобразования этих явно связанных типов (в предположении о том, что такое преобразование имеет смысл).

Конечно, проблему можно решить с помощью создания в этих в структурах вспомогательных методов (например, Rectangle.ToSquare()), но в C# можно создавать пользовательские подпрограммы преобразования, позволяющие соответствующим типам по-своему отвечать на операцию (). Так, при правильной конфигурации типа Square вы получите возможность использовать следующий синтаксис для явного преобразования этих типов структуры.

**// Превращение прямоугольника в квадрат.**

Rectangle rect;

rect.Width = 3;

rect.Height = 10;

Square sq = (Square)rect;

**Создание пользовательских подпрограмм преобразования**

В C# есть два ключевых слова, explicit и implicit, предназначенные для управления тем, как типы должны отвечать на попытки преобразования. Предположим, что у нас есть следующие определения структур.

public struct Rectangle {

**// Открыты для простоты,**

**// но ничто не мешает инкапсулировать их в виде свойств.**

 public int Width, Height;

 public void Draw() { Console.WriteLine("Отображение прямоугольника."); }

 public override string ToString() {

  return string.Format("[Ширина = {0}; Высота = {1}]", Width, Height);

 }

}

public struct Square {

 public int Length;

 public void Draw() { Console.WriteLine("Отображение квадрата."); }

 public override string ToString() { return string.Format("[Сторона = {0}]", Length); }

**// Rectangle (прямоугольник) можно явно преобразовать**

**// в Square (квадрат)**.

 public static explicit operator Square(Rectangle r) {

  Square s;

  s.Length = r.Width;

  return s;

 }

}

Обратите внимание на то, что на этот раз для типа Reсtangle определяется операция явного преобразования. Как и при перегрузке встроенных операций, в C# для подпрограмм преобразования используется ключевое слово operator (в совокупности с ключевым словом explicit или implicit) и эти подпрограммы должны определяться, как статические. Входным параметром является объект, который вы хотите преобразовать, а возвращаемое значение – это объект, в который поступающий объект превращается.

public static explicit operator Square(Rectangle r) {…}

Здесь предполагается, что квадрат (который является геометрической фигурой с равными сторонами) можно получить на основе ширины прямоугольника. Поэтому вы можете превратить Rectangle (прямоугольник) в Square (квадрат) так.

static void Main(string args) {

 Console.WriteLine("\*\*\*\*\* Забавы с преобразованиями \*\*\*\*\*\n");

**// Создание прямоугольника 10 х 5.**

 Rectangle rect;

 reсt.Width = 10;

 rect.Height = 5;

 Console.WriteLine("rect = {0}", rect);

**// Преобразование прямоугольника в квадрат 10 х 10.**

 Square sq = (Square)rect*;*

 Console.WriteLine("sq = {0}", sq);

 Console.ReadLine();

}

Наверное, от превращения прямоугольников в квадраты в рамках одного контекста не слишком много пользы, но предположим, что у нас есть функция, которая предполагает использование типов Square.

**// Этот метод требует использования типа Square.**

private static void DrawSquare(Square sq) {

 sq.Draw();

}

Используя нашу операцию явного преобразования, мы можем передавать этой функции типы Square.

static void Main(string[] args) {

 …

**// Преобразование Rectangle в Square для вызова метода.**

 DrawSquare((Square)rect);

}

**Варианты явного преобразования для типа Square**

Теперь вы можете явно превращать прямоугольники в квадраты, но рассмотрим еще несколько вариантов явного преобразования. Поскольку у квадрата стороны равны, можно явно преобразовать System.Int32 в Square (длина стороны квадрата будет равна значению поступающего целого числа). Аналогично можно изменить определение Square, если требуется обеспечить преобразование из Square в System.Int32. Вот логика соответствующего вызова.

static void Main(string[] args) {

 …

**// Преобразование System.Int32 в Square.**

 Square sq2 = (Square)90;

 Console.WriteLine("sq2 = {0}", sq2);

**// Преобразование Square в System.Int32.**

 int side = (int)sq2;

 Console.WriteLine("Длина стороны sq2 = {0}", side);

}

А вот как следует обновить определение типа Square.

public struct Square {

 …

 public static **explicit**operator Square(int sideLength) {

  Square newSq;

  newSq.Length = sideLength;

  return newSq;

 }

 public static **explicit**operator int(Square s) { return s.Length; }

}

Выглядит немного странно, не так ли? Честно говоря, преобразование из Square в System.Int32 не является интуитивно очевидной (или полезной) операцией. Однако она демонстрирует одну очень важную особенность пользовательских подпрограмм преобразования; компилятору "все равно" из чего и во что вы преобразуете – важно, чтобы ваш программный код был синтаксически правильным. Так что, как в случае с перегрузкой операций, только из того, что вы можете создать операцию явного преобразования для данного типа, совсем не следует, что вы обязаны это делать. Как правило, этот подход оказывается наиболее полезным тогда, когда создаются типы структуры .NET, поскольку такие типы не могут использовать иерархии классического наследования (для которых соответствующие преобразования реализуются автоматически).

**Определение подпрограмм неявного преобразования**

До этого момента мы с вами создавали пользовательские операции явного преобразования. Но что можно сказать о следующем неявном преобразовании?

static void Main(string[] args) {

 …

**// Попытка выполнить неявное преобразование?**

 Square s3;

 s3.Length = 83;

 Rectangle rect2 = s3;

}

Как вы можете догадаться сами, этот программный код скомпилирован не будет, поскольку в нем не предлагается никакой подпрограммы неявного преобразования для типа Rectangle. Тут нас подстерегает "ловушка": в одном и том же типе нельзя определять явные и неявные функции преобразования, не отличающиеся по типу возвращаемого значения или по набору параметров. Может показаться, что это правило является слишком ограничивающим, но не следует забывать о том, что даже если тип определяет подпрограмму неявного преобразования, вызывающая сторона "имеет право" использовать синтаксис явного преобразования!

Запутались? Чтобы прояснить ситуацию, добавим в структуру Rectangle подпрограмму неявного преобразования, используя ключевое слово C# implicit (в следующем программном коде предполагается, что ширина результирующего Rectangle получается с помощью умножения стороны Square на 2).

public struct Rесtangle {

 …

 public static implicit operator Rectangle(Square s) {

  Rectangle r;

  r.Height = s.Length;

**// Ширина нового прямоугольника равна**

**// удвоенной длине стороны квадрата.**

  r.Width = s.Length \* 2;

 }

}

С такими изменениями вы получаете возможность преобразовывать указанные типы так.

static void Main(string[] args) {

 …

**// Неявное преобразование: все OK!**

 Square s3;

 s3.Length = 83;

 Rectangle rect2 = s3;

 Console.WriteLine("rect2 = {0}", rect2);

 DrawSquare(s3);

**// Синтаксис явного преобразования: тоже OK!**

 Square s4;

 S4.Length = 3;

 Rectangle rect3 = (Rectangle)s4;

 Console.WriteLine("rect3 = {0}", rect3);

 …

}

Снова подчеркнем, что допускается определение подпрограмм и явного, и неявного преобразования для одного и того же типа, но только если отличаются их сигнатуры. Поэтому мы можем обновить Square так, как показано ниже.

public struct Square {

 …

**// Можно вызывать как Square sq2 = (Square)90;**

**// или как Square sq2**= **90;**

 public static implicit operator Square(int sideLength) {

  Square newSq;

  newSq.Length = sideLength;

  return newSq;

**// Должно вызываться как int side = (Square)mySquare;**

  public static explicit operator int(Square s) { return s.Length; }

 }

}

**Внутреннее представление пользовательских подпрограмм преобразования**

Как и в случае перегруженных операций, те методы, которые обозначены ключевыми словами implicit или explicit, получают "специальные имена" в терминах CIL: op\_Implicit и op\_Explicit соответственно (рис. 9.2).

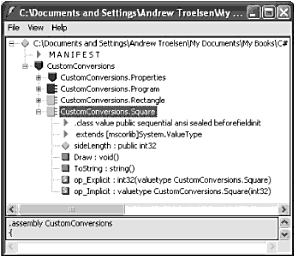


Рис. 9.2. Представление пользовательских подпрограмм преобразования в терминах CIL.

На этом мы завершаем обзор возможностей пользовательских подпрограмм преобразования. Как и в случае перегруженных операций, соответствующий синтаксис является лишь сокращённым вариантом определения "нормальных" членов-функций, и с этой точки зрения он не является обязательным.

**Исходный код.**Проект CustomConversions размещен в подкаталоге, соответствующем главе *9.*

**Ключевые слова C#, предназначенные для более сложных конструкций**

В завершение главы мы рассмотрим ряд ключевых слов C#, применение которых требует от разработчика несколько большего опыта в программировании:

• checked/unchecked;

• unsafe/stackalloc/fixed/sizeof.

Сначала мы выясним, как с помощью ключевых слов checked и unchecked в C# обеспечивается автоматическое выявление условий переполнения и потери значимости при выполнении арифметических операций.

**Ключевое слово checked**

Вы, несомненно, прекрасно знаете, что любой числовой тип данных имеет свои строго заданные верхний и нижний пределы (значения которых можно выяснить программными средствами с помощью свойств MaxValue и MinValue). При выполнении арифметических операций с конкретным типом вполне возможно случайное *переполнение*блока хранения данного типа (попытка присвоения типу значения, которое оказывается больше максимально допустимого) или *потеря значимости*(попытка присвоения значения, которое оказывается меньше минимально допустимого). Чтобы "идти в ногу" с CLR, обе эти возможности будут обозначаться, как "переполнение". (И переполнение, и потеря значимости приводят к созданию типа System.OverflowException. Типа System.UnderflowException в библиотеках базовых классов нет.)

Для примера предположим, что мы создали два экземпляра типа System.Byte (тип byte в C#), присвоив им значения, не превышающие максимального (255). При сложении значений этих типов (с условием преобразования результата в тип byte) хотелось бы предполагать, что результат будет точной суммой соответствующих членов.

namespace CheckedUnchecked {

 class Program {

  static void Main(string[] args) {

**// Переполнение для System.Byte.**

   Console.WriteLine("Макс, значение для byte равно {0}", byte.MaxValue);

   Console.WriteLine("Мин. значение для byte равно {0}", byte.MinValue);

   byte b1 = 100;

   byte b2 = 250;

   byte sum = (byte)(b1 + b2);

**// Значением sum должно быть 350, но.…**

   Console.WriteLine("sum = {0}", sum);

   Console.ReadLine();

  }

 }

}

Вывод этого приложения покажет, что sum содержит значение 94 (а не ожидаемое 350). Причина очень проста. Поскольку System.Byte может содержать только значения, находящиеся между 0 и 255 (что в итоге составляет 256 значений), sum будет содержать значение переполнения (350 – 256 = 94). Как видите, в отсутствие специальной коррекции переполнение происходит без генерирования исключений. Иногда скрытое переполнение не создает никаких проблем. В других случаях соответствующая потеря данных может быть совершенно неприемлемой.

Для обработки переполнений или потери значимости в приложении имеются две возможности. Первой возможностью является использование программистского опыта и квалификации с тем, чтобы обработать все условия переполнения вручную. Предполагая, что вы можете найти все условия переполнения в программе, можно было бы решить проблему, связанную с переполнением в предыдущем программном коде, как показано ниже.

**// Использование int для sum, чтобы не допустить переполнения.**

byte b1 = 100;

byte b2 = 250;

**int**sum = b1 + b2;

Конечно, проблемой этого подхода является то, что вы – человек, а значит, при всех ваших усилиях, могут остаться ошибки, ускользнувшие от вашего взгляда. Поэтому в C# предлагается ключевое слово checked. При помещении оператора (или блока операторов) в рамки контекста ключевого слова checked компилятор C# генерирует специальные CIL-инструкщии, с помощью которых проверяются условия переполнения, возможные при выполнении сложения, умножение, вычитания или деления числовых типов данных. Если происходит переполнение, среда выполнения генерирует тип System.OverflowException. Рассмотрите следующую модификацию программы.

class Program {

 static void Main(string[] args) {

**// Переполнение для System.Byte.**

  Console.WriteLine("Макс. значение для byte равно {0}.", byte.MaxValue);

  byte b1 = 100;

  byte b2 = 250;

  try {

   byte sum = **checked**((byte)(b1 + b2));

   Console.WriteLine("sum = {0}", sum);

  } catch (OverflowException e) { Console.WriteLine(e.Message); }

 }

}

Здесь оператор сложения b1 и b2 помещается в контекст ключевого слова checked. Если вы хотите, чтобы проверка переполнения происходила для блока программного кода, можно взаимодействовать с ключевым словом checked так, как показано ниже.

try {

**checked**{

  byte sum = (byte)(b1 + b2);

  Console.WritaLine(sum = {0}", sum);

 }

} catch (OverflowException e) {

 Console.WriteLine(e.Message);

}

В любом случае соответствующий программный код будет проверяться на возможное переполнение автоматически, и если переполнение будет обнаружено, то будет сгенерирована соответствующее переполнению исключение.

**Проверки переполнения для всего проекта**

Если вы создаете приложение, которое не должно позволять скрытое переполнение ни при каких условиях, будет слишком утомительно указывать ключевое слово checked для каждой строки программного кода. В качестве альтернативы компилятор C# предлагает использовать флаг /checked. Когда этот флаг активизирован, *все*арифметические операции будут проверяться на переполнение без указания ключевого слова checked. Если обнаружится переполнение, вы получите OverflowException среды выполнения.

Чтобы активизировать этот флаг в Visual Studio 2005, откройте страницу свойств проекта и щелкните на кнопке Advanced на вкладке Build. В появившемся диалоговом окне отметьте флажок Check for arithmetic overflow/underflow (Проверять условия переполнения/потери значимости для арифметических операций), рис. 9.3.

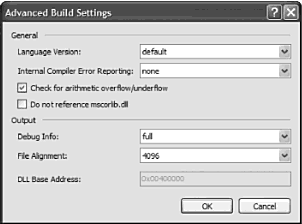


Рис. 9.З. Активизация проверки переполнения в Visual Studio 2005

Ясно, что эта установка оказывается очень полезной при отладке. После того как все связанные с переполнением исключения будут из программного кода удалены, флаг /checked для последующей компоновки можно отключить.

**Ключевое слово unchecked**

В предположении, что вы активизировали проверку переполнения для всего проекта, как разрешить игнорирование переполнений для тех блоков программного кода, где "молчаливая реакция" на переполнение вполне приемлема? Поскольку флаг /checked предполагает проверку *всей*арифметической логики, в языке C# предлагается ключевое слово unchecked, которое позволяет отключить генерирование System.OverflowException для конкретных случаев, Правила использования этого ключевого слова аналогичны правилам использования ключевого слова checked, и вы можете указать для него один оператор или блок операторов, например:

**// Даже если флаг /checked активизирован,**

**// этот блок не генерирует исключения в среде выполнения.**

unchecked {

 byte sum = (byte)(b1 + b2);

 Console.WriteLine(sum = {0}", sum)*;*

}

Подводя итога обсуждения ключевых слов C# checked и unchecked, снова подчеркнем, что по умолчанию среда выполнения .NET игнорирует условии переполнения, возникающие при использовании арифметических операций. Если вы хотите селективно контролировать условия переполнения для отдельных операторов, используйте ключевое слово checked. Если нужно контролировать ошибки переполнения во всем приложении, укажите флаг /checked, Наконец, можно использовать ключевое слово unchecked, если у вас есть блок программного кода, для которого переполнение приемлемо (и поэтому оно не должно генерировать исключение в среде выполнения).

Исходный код. Проект CheckedUnchecked размещен в подкаталоге, соответствующем главе 9.

**Работа с типами указателя**

Из главы 8 вы узнали, что платформа .NET определяет две главные категории данных: типы, характеризуемые значениями, и типы, характеризуемые ссылками (ссылочные типы). Однако, справедливости ради., следует сказать, что имеется и третья категория: это *типы указателя.*Для работы с типами указателя предлагаются специальные операции и ключевые слова, с помощью которых можно "обойти" схему управления памятью CLR и "взять управление в свои руки" (табл. 9.3).

**Таблица 9.3.**Операции и ключевые слова C# для работы с указателями

|  |  |
| --- | --- |
| **Операция или ключевое слово** | **Описание** |
| \* | Используется для создания *переменной указателя*(т.е. переменной, представляющей непосредственно адресуемую точку в памяти). Как и в C(++), тот же знак используется для операции разыменования указателя (т.е, для операции, которая возвратит значение, размещенное по адресу, указанному операндом) |
| & | Используется для получения адреса переменной в памяти |
| –› | Используется для доступа к полям типа, представленным указателем (небезопасная версия операции, обозначаемой в C# точкой) |
| [] | Операция [] (в небезопасном контексте) позволяет индексировать элемент, на который указывает переменная указателя. (Обратите внимание на аналогию между переменной указателя и операцией [] в C(++).) |
| ++, -- | В небезопасном контексте к типам указателя могут применяться операции приращения и отрицательного приращения |
| +, - | В небезопасном контексте к типам указателя могут применяться операции сложения и вычитания |
| ==, !=, <, >, <=, >= | В небезопасном контексте к типам указателя могут применяться операции сравнения и проверки на тождественность |
| stackalloc | В небезопасном контексте можно использовать ключевое слово stackalloc, чтобы размещать массивы C# в стеке |
| fixed | В небезопасном контексте можно использовать ключевое слово fixed*,*временно фиксирующее переменную с тем, чтобы можно было найти ее адрес |

Перед рассмотрением деталей позвольте заметить, что необходимость в использовании типов указателя возникает *очень редко,*если она возникает вообще. Хотя C# и позволяет "спуститься" на уровень манипуляций с указателями, следует понимать, что среда выполнения .NET не имеет никакого представления о ваших намерениях. Поэтому если вы ошибетесь в направлении указателя, то за последствия будете отвечать сами. Если учитывать это, то когда же на самом деле возникает необходимость использования типов указателя? Есть две стандартные ситуации.

• Вы стремитесь оптимизировать работу определенных частей своего приложения путем непосредственной манипуляции памятью вне пределов управления CLR.

• Вы хотите использовать методы C-библиотеки \*.dll или COM-сервера, требующие ввода указателей в виде параметров.

Если вы решите использовать указанную возможность языка C#, необходимо информировать csc.exe об этих намерениях, указав разрешение для проекта поддерживать "небезопасный программный код". Чтобы сделать это с командной строки компилятора C# (csc.exe), просто укажите в качестве аргумента флаг /unsafe. В Visual Studio 2005 вы должны перейти на страницу свойств проекта и активизировать опцию Allow Unsafe Code (Разрешать использование небезопасного программного кода) на вкладке Build (рис. 9.4).

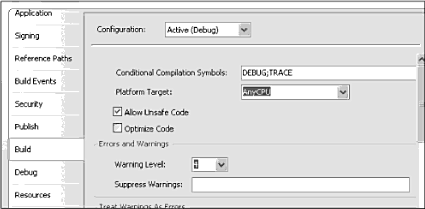


Рис. 9.4. Разрешение небезопасного программного кода Visual Studio 2005

**Ключевое слово unsafe**

В следующих примерах предполагается, что вы имеете опыт работы с указателями в C(++). Если это не так, не слишком отчаивайтесь. Подчеркнем еще раз, что создание небезопасного программного кода не является типичной задачей в большинстве приложений .NET. Если вы хотите работать с указателями в C#, то должны специально объявить блок программного кода "небезопасным", используя для этого ключевое слово unsafe (как вы можете догадаться сами, весь программный код, не обозначенный ключевым unsafe, автоматически считается "безопасным").

**unsafe**{

**// Операторы для работы с указателями.**

}

Кроме объявления контекста небезопасного программного кода, вы можете строить "небезопасные" структуры, классы, члены типов и параметры. Вот несколько примеров, на которые следует обратить внимание.

**// Вся эта структура является 'небезопасной'**

**// и может использоваться только в небезопасном контексте.**

public **unsafe**struct Node {

 public int Value;

 public Node\* Left;

 public Node\* Right;

}

**// Эта структура является безопасной, но члены Node\* – нет.**

**// Строго говоря, получить доступ к 'Value' извне небезопасного**

**// контекста можно, а к 'Left' и 'Right'**- **нет.**

public struct Node {

 public int Value;

**// К этим элементам можно получить доступ только**

**// в небезопасном контексте!**

 public **unsafe**Node\* Left;

 public **unsafe**Node\* Right;

}

Методы (как статические, так и уровня экземпляра) тоже можно обозначать, как небезопасные. Предположим, например, вы знаете, что некоторый статический метод использует логику указателей. Чтобы гарантировать вызов данного метода только в небезопасном контексте, можно определить метод так, как показано ниже.

**unsafe**public static void SomeUnsafeCode() {

**// Операторы для работы о указателями.**

}

В такой конфигурации требуется, чтобы вызывающая сторона обращалась к SomeUnsafeCode() так.

static void Main(string[] args) {

 unsafe {

  SomeUnsafeCode();

 }

}

Если же не обязательно, чтобы вызывающая сторона делала вызов в небезопасном контексте, то можно не указывать ключевое слово unsafe в методе SomeUnsafeCode() и записать следующее:

public static void SomeUnsafeCode() {

 unsafe {

**// Операторы для работы с указателями.**

 }

}

что должно упростить вызов:

static void Main(string[] args) {

 SomeUnsafeCode();

}

**Работа с операциями \* и &**

После создания небезопасного контекста вы можете строить указатели на типы с помощью операции \* и получать адреса заданных указателей с помощью операции &. В C# операция \* применяется только к соответствующему типу, а не как префикс ко всем именам переменных указателя. Например, в следующем фрагменте программного кода объявляются две переменные типа int\* (указатель на целое).

**// Нет! В C# это некорректно!**

int \*pi, \*pj;

**// Да! Это в C# правильно.**

int\* pi, pj;

Рассмотрим следующий пример.

unsafe {

 int myInt;

**// Определения указателя типа int**

**// и присваивание ему адреса myInt.**

 int\* ptrToMyInt = &myInt;

**// Присваивание значения myInt**

**// с помощью разыменования указателя**.

 \*ptrToMyInt = 123;

**// Печать статистики.**

 Console.WriteLine("Значение myInt {0}", myInt);

 Console.WriteLine("Адрес myInt {0:X}", (int)&ptrToMyInt);

}

**Небезопасная (и безопасная) функция Swap**

Конечно, объявление указателей на локальные переменные с помощью просто-то присваивания им значений (как в предыдущем примере) никогда не требуется. Чтобы привести более полезный пример небезопасного программного кода, предположим, что вы хотите создать функцию обмена, используя арифметику указателей.

unsafe public static void UnsafeSwap(int\* i, int\* j) {

 int temp = \*i;

 \*i = \*j;

 \*j = temp;

}

Очень похоже на C, не так ли? Однако с учетом знаний, полученных из главы 3, вы должны знать, что можно записать следующую безопасную версию алгоритма обмена, используя ключевое слово C# ref.

public static void SafeSwap(ref int i, ref int j)

 int temp = i;

 i = j;

 j = temp;

}

Функциональные возможности каждого из этих методов идентичны, и это еще раз подтверждает, что работа напрямую с указателями в C# требуется редко. Ниже показана логика вызова,

static void Main(string[] args) {

 Console.WriteLine(\*\*\* Вызов метода с небезопасным кодом \*\*\*");

**// Значения для обмена.**

 int i = 10, i = 20;

**// 'Безопасный' обмен значениями.**

 Console.WriteLine("\n\*\*\*\*\* Безопасный обмен \*\*\*\*\*");

 Cоnsоle.WriteLine("Значения до обмена: i = {0}, j = {1}", i, j);

 SafeSwap(ref 1, ref j);

 Console.WriteLine("Значения после обмена: i = {0}, j = {l}", i*,*j);

**// 'Небезопасный' обмен значениями.**

 Console.WriteLine("\n\*\*\*\*\* Небезопасный обмен \*\*\*\*\*");

 Console.WriteLine("Значения до обмена: i = {0}, j = {1}", i, j);

 unsafe { UnsafeSwap(&i, &j); }

 Console.WriteLine("Значения после обмена: i = {0}, j = {1}", i, j);

 Console.ReadLine();

}

**Доступ к полям через указатели (операция -›)**

Теперь предположим, что у нас определена структура Point и мы хотим объявить указатель на тип Point. Как и в C(++), для вызова методов или получения доступа к полям типа указателя необходимо использовать операцию доступа к полю указателя (-›). Как уже упоминалось в табл. 9.3, это небезопасная версия стандартной (безопасной) операции, обозначаемой точкой (.). Фактически, используя операцию разыменования указателя (\*). можно снять косвенность указателя, чтобы (снова) вернуться к применению нотации, обозначаемой точкой. Рассмотрите следующий программный код.

struct Point {

 public int x;

 public int y;

 public override string ToString() { return string.Format ("({0}, {1})"*,*x, y); }

}

static void Main(string[] args) {

**// Доступ к членам через указатели.**

 unsafe {

  Point point;

  Point\* p =&point;

  p-›x = 100;

  p-›y = 200;

  Console.WriteLine(p-›ToString());

 }

**// Доступ к членам через разыменование указателей.**

 unsafe {

  Point point;

  Point\* p =&point;

  (\*p).x = 100;

  (\*p).y = 200;

  Console.WriteLine((\*p).ToString());

 }

}

**Ключевое слово stackalloc**

В небезопасном контексте может понадобиться объявление локальной переменной, размещаемой непосредственно в памяти стека вызовов (и таким образом не подлежащей "утилизации" при сборке мусора .NET). Чтобы сделать такое объявление, в C# предлагается ключевое слово stackalloc являющееся C#-эквивалентом функции alloca из библиотеки времени выполнения C. Вот простой пример.

unsafe {

 char\* p = **stackalloc**char[256];

 for (int k = 0; k ‹ 256; k++) p[k] = (char)k;

}

**Фиксация типа с помощью ключевого слова fixed**

Как показывает предыдущий пример, задачу размещения элемента в памяти в рамках небезопасного контекста можно упростить с помощью ключевого слова stackalloc. В силу самой природы этой операции соответствующая память очищается, как только происходит возврат из метода размещения (поскольку память выбирается из стека). Но рассмотрим более сложный пример. В процессе обсуждения операции -› вы создали характеризуемый значением тип Point. Подобно всем типам, характеризуемым значениями, выделенная для него память в стеке освобождается сразу же после исчезновения контекста выполнения. Теперь, для примера, предположим что тип Point был определен как ссылочный тип.

**class** Point { **//‹= Теперь это класс!**

 public int x;

 public int у;

 public override string ToString() { return string.Format("({0}, {1})", x, y); }

}

Вы хорошо знаете о том, что если вызывающая сторона объявляет переменную типа Point, для нее выделяется динамическая память, являющаяся объектом для сборки мусора. Тогда возникает вопрос: что произойдет, если небезопасный контекст попытается взаимодействовать с соответствующим объектом (или любым другим объектом в динамической памяти)? Поскольку сборка мусора может начаться в любой момент, представьте себе всю болезненность доступа к членам Point, когда идет очистка динамической памяти. Теоретически возможно, что небезопасный контекст будет пытаться взаимодействовать с членом, который больше не доступен или занимает новое положение в динамической памяти, "пережив" очередную генерацию чистки (и это, очевидно, является проблемой).

Чтобы блокировать переменную ссылочного типа в памяти из небезопасного контекста, в C# предлагается ключевое слово fixed. Оператор fixed устанавливает указатель на управляемый тип и "закрепляет" переменную на время выполнения оператора. Без ключевого слова fixed в применении указателей на управляемые переменные было бы мало смысла, поскольку в результате сборки мусора. такие переменные могут перемещаться непредсказуемым образом. (На самом деле компилятор C# вообще не позволит установить указатель на управляемую переменную, если в операторе не используется ключевое слово fixed.)

Так что, если вы создадите тип Point (сейчас переопределенный, как класс) и захотите взаимодействовать с его членами, то должны записать следующий программный код (иначе возникнет ошибка компиляции).

unsafe public static void Main() {

 point pt = new Point();

 pt.x = 5;

 pt.y = 6;

**// Фиксация pt, чтобы не допустить перемещения**

**// или удаления при сборке мусора.**

**fixed**(int\* p =&pt.x) {

**// Переменная int\* используется здесь.**

 }

**// Теперь pt не зафиксирована и может быть убрана**

**// сборщиком мусора.**

 Console.WriteLine("Значение Point: {0}", pt);

}

В сущности, ключевое слово fixed позволяет строить операторы, закрепляющие ссылочную переменную в памяти, чтобы ее адрес оставался постоянным на время выполнения оператора. Для гарантии безопасности обязательно фиксируйте ссылки при взаимодействии со ссылочными типами из небезопасного контекста программного кода.

**Ключевое слово sizeof**

В заключение обсуждения вопросов, связанных с небезопасным контекстом в C#, рассмотрим ключевое слово sizeof. Как и в C(++), ключевое слово C# sizeof используется для того, чтобы выяснить размер в байтах типа, характеризуемого значениями (но не ссылочного типа), и это ключевое слово может использоваться только в рамках небезопасного контекста. Очевидно, что указанная возможность может оказаться полезной при взаимодействии с неуправляемыми API, созданными на базе C. Использовать ее очень просто.

unsafe {

 Console.WriteLine("Длина short равна {0}.", **sizeof**(short));

 Console.WriteLine("Длина int равна {0}.", **sizeof**(int));

 Console.WriteLine("Длина long равна {0}.", **sizeof**(long));

}

Поскольку sizeof может оценить число байтов для любого элемента, производного от System.ValueType, можно получать размеры пользовательских структур. Допустим, мы определили следующую структуру.

struct MyValueType {

 public short s;

 public int i;

 public long l;

}

Тогда ее размеры можно выяснить так.

unsafe {

 Console.WriteLine("Длина short равна {0}.", **sizeof**(short));

 Console.WriteLine("Длина int равна {0}.", **sizeof**(int));

 Console.WriteLine("Длина long равна {0}.", **sizeof**(long));

 Console.WriteLine("Длина MyValueType равна {0}."/ **sizeof**(MyValueType));

}

**Исходный код.**Проект UnsafeCode размещен в подкаталоге, соответствующем главе 9.

**Директивы препроцессора C#**

Подобно многим другим языкам из семейства C, в C# поддерживаются различные символы, позволяющие влиять на процесс компиляции. Перед рассмотрением директив препроцессора C# согласуем соответствующую терминологию. Термин "директива препроцессора C#" не вполне точен. Фактически этот термин используется только для согласованности с языками программирования C и C++. В C# нет отдельного шага препроцессора. Директивы препроцессора в C# являются составной частью процесса лексического анализа компилятора.

Так или иначе, синтаксис директив препроцессора C# очень похож на синтаксис соответствующих директив остальных членов семейства C в том, что эти директивы всегда имеют префикс, обозначенный знаком "диез" (#). В табл. 9.4 описаны некоторые из наиболее часто используемых директив (подробности можно найти в документации .NET Framework 2.0 SDK).

Таблица 9.4. Типичные директивы препроцессора C#

|  |  |
| --- | --- |
| **Директивы** | **Описание** |
| #region, #endregion | Используются для обозначения разделов стягиваемого исходного кода |
| #define, #undef | Используются для определения и отмены определения символов условной компиляции |
| #if, #elif, #else, #endif | Используются для условного пропуска разделов исходного кода (на основе указанных символов компиляции) |

**Разделы программного кода**

Возможно, одной из самых полезных директив препроцессора являются #region и #endregion. Используя эти признаки, вы указываете блок программного кода, который можно скрыть от просмотра и идентифицировать информирующим текстовым маркером. Использование разделов программного кода может упростить обслуживание больших файлов \*.cs. Можно, например, создать один раздел для конструкторов типа, другой – для свойств и т.д.

class Car {

 private string petName;

 private int currSp;

#**region Constructors**

 public Car() {…}

 public Car Car(int currSp, string petName) {…}

**#endregion**

**#region Properties**

 public int Speed {…}

 public string Name {…}

**#endregion**

}

При помещений указателя мыши на маркер свернутого раздела вы получите снимок программного кода, спрятанного за соответствующим названием (рис. 9.5).

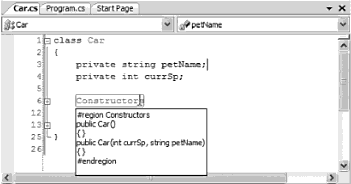


Рис. 9.5. Разделы программного кода за работой

**Условная компиляция**

Другой пакет директив препроцессора (#if, #elif, #else, #endif) позволяет выполнить компиляцию блока программного кода по условию, базируясь на предварительно заданных символах. Классическим вариантом использования этих директив является идентификация блока программного кода, который компилируется только при отладке (а не при окончательной компоновке).

class Program

 static void Main(string[] args) {

**// Этот программный код выполняется только при отладочной**

**// компиляции проекта.**

#if DEBUG

  Console.WriteLine("Каталог приложения: {0}", Environment.CurrentDirectory);

  Console.WriteLine("Блок: {0}", Environment.MachineName);

  Console.WriteLine("ОС: {0}", Environment.OSVersion);

  Console.WriteLine("Версия .NET: {0}", Environment.Version);

#endif

 }

}

Здесь выполняется проверка на символ DEBUG. Если он присутствует, выводится ряд данных состояния, для чего используются соответствующие статические члены класса System.Environment. Если символ DEBUG не обнаружен, то программный код, размещенный между #if и #endif, компилироваться не будет и в результирующий компоновочный блок не войдет, т.е. будет фактически проигнорирован.

По умолчанию Visual Studio 2005 всегда определяет символ DEBUG, однако такое поведение можно отменить путем снятия отметки флажка Define DEBUG constant (Определить константу DEBUG) на вкладке Build (Сборка), размещенной на странице Properties (Свойства) вашего проекта. В предположении о том, что этот обычно генерируемый символ DEBUG отключен, можно определить этот символ для*каждого*файла в отдельности, используя директиву препроцессора #define.

#define DEBUG using System;

namespace Preprocessor {

 class ProcessMe {

  static void Main(string[] args) {

**// Программный код, подобный показанному выше…**

}

 }

}

**Замечание**. Директивы #define в файле с программным кодом C# должны быть указаны до всех остальных.

Можно также определять свои собственные символы препроцессора. Предположим, например, что у нас есть класс C#, которой должен компилироваться немного иначе в рамках дистрибутива Mono.NET (см. главу 1). Используя #define, можно определить символ MONO\_BUILD для каждого файла.

#define DEBUG

#define MONO\_BUILD

using System;

namespace Preprocessor {

 class Program {

  static void Main (string[] args) {

#if MONO\_BUILD

   Console.WriteLine("Компиляция для Mono!");

*#*else

   Consоlе.WriteLine("Компиляция для Microsoft .NET");

#endif

  }

 }

}

Чтобы создать символ, применимый для всего проекта, используйте текстовый блок Conditional compilation symbols (Символы условной компиляции, размещенный на вкладке Build (Сборка) страницы свойств проекта (рис. 9.6).

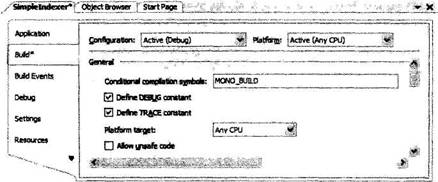


Рис. 9.6. Определение символа препроцессора для применения в рамках всего проекта

**Резюме**

Целью этой главы является более глубокое изучение возможностей языка программирования C#. Глава началась с обсуждения ряда достаточно сложных конструкций программирования (методов индексатора, перегруженных операций и пользовательских подпрограмм преобразования). Затем был рассмотрен небольшой набор не слишком широко известных ключевых слов (таких, как sizeof, checked, unsafe и т.д.), обсуждение которых естественно привело к рассмотрению вопросов непосредственной работы с типами указателя. При исследовании типов указателя было показано, что в подавляющем большинстве приложений C# для использования типов указателя нет никакой необходимости.

**ГЛАВА 10. Обобщения**

С появлением .NET 2.0 язык программирования C# стал поддерживать новую возможность CTS (Common Type System – общая система типов), названную *обобщениями*(generics). Упрощенно говоря, обобщения обеспечивают программисту возможность определения "заполнителей" (формально называемых *параметрами типа)*для аргументов методов и определений типов, которые будут конкретизированы во время вызова обобщенного метода или при создании обобщенного типа.

С целью иллюстрации этой новой возможности языка мы начнем главу с рассмотрения пространства имен System.Collections.Generic. Рассмотрев примеры поддержки обобщений в библиотеках базовых классов, в остальной части этой главы мы попытаемся выяснить, как строить свои собственные обобщения членов, классов, структур, интерфейсов и делегатов.

**Снова о создании объектных образов, восстановлении значений и System.Object**

Чтобы понять, в чем заключаются преимущества использования обобщений, следует выяснить, какие проблемы возникают у программиста без их использования. Вы должны помнить из главы 3, что платформа .NET поддерживает автоматическое преобразование объектов стека и динамической памяти с помощью операций создания объектного образа и восстановления из объектного образа. На первый взгляд, эта возможность кажется не слишком полезной и имеющей, скорее, теоретический, чем практический интерес. Но на самом деле возможность создания объектного образа и восстановления из объектного образа очень полезна тем, что она позволяет интерпретировать все, как System.Object, оставив все заботы о распределении памяти на усмотрение CLR.

Чтобы рассмотреть особенности процесса создания объектного образа, предположим, что мы создали System.Collections.ArrayList для хранения числовых (т.е. размещаемых в стеке) данных. Напомним, что все члены ArrayList обладают прототипами для получения и возвращения типов System.Object. Но вместо того, чтобы заставлять программиста вручную вкладывать размещенное в стеке целое число в соответствующую объектную оболочку, среда выполнения делает это автоматически с помощью операции создания объектного образа.

static void Main(string[] args) {

**// При передаче данных члену, требующему объект, для**

**// характеризуемых значениями типов автоматически создается**

**// объектный образ.**

 ArrayList myInts = new ArrayList();

 myInts.Add(10)**;**

 Console.ReadLine();

}

Чтобы восстановить значение из объекта ArrayList, используя индексатор типа, вы должны превратить размещенный в динамической памяти объект в размещенное в стеке целое число, используя операцию преобразования.

static void Main(string[] args) {

 …

**// Значение восстанавливается… и снова становится объектом!**

 Console.WriteLine("Значение вашего int: {0}", (int)myInts[0]);

 Console.ReadLine();

}

Для представления операции создания объектного образа в терминах CIL компилятор C# использует блок box. Точно так же операция восстановления из объектного образа преобразуется в CIL-блок unbox. Вот соответствующий CIL-код для показанного выше метода Main() (этот код можно увидеть с помощью ildasm.exe).

.method private hidebysig static void Main(string[] args) cil managed {

 …

**box [mscorlib]System.Int32**

 callvirt instance int32 [mscorlib] System.Collections.ArrayList::Add(object)

 pop

 ldstr "Значение вашего int: {0}"

 ldloc.0

 ldc.i4.0

 callvirt instance object [mscorlib] System.Collections.ArrayList::get\_Item(int32)

**unbox [mscorlib]System.Int32**

ldind.i4

**box [mscorlib]System.Int32**

call void [mscorlib]System.Console::WriteLine(string, object)

 …

}

Обратите внимание на то. что перед обращением к ArrayList.Add() размещенное в стеке значение System.Int32 преобразуется в объект, чтобы передать требуемый System.Object. Также заметьте, что при чтении из ArrayList с помощью индексатора типа (что отображается в скрытый метод get\_Item()) объект System.Object восстанавливается в System.Int32 только для того, чтобы снова стать объектным образом при передаче методу Console.WriteLine().

**Проблемы создания объектных образов и восстановления значений**

Операции создания объектных образов и восстановления из них значений очень удобны с точки зрении программиста, но такой упрощенный подход при обмене элементами стека и динамической памяти имеет свои специфические проблемы производительности и не гарантирует типовой безопасности. Чтобы понять проблемы производительности, рассмотрим следующие шаги, которые приходится выполнять при создании объектного образа и восстановлении значения обычного целого числа.

1. Новый объект нужно разместить в управляемой динамической памяти.

2. Значение размещенных в стеке данных нужно записать в соответствующее место в памяти.

3. При восстановлении значений, сохраненного в объекте, размещенном в динамической памяти, это значение нужно снова вернуть в стек.

4. Неиспользуемый объект в управляемой динамической памяти (в конце концов) должен быть уничтожен сборщиком мусора.

В нашей ситуации метод Main() с точки зрения производительности не имеет никаких проблем, но соответствующие проблемы появятся, когда ArrayList будет содержать тысячи целых значений, если вашей программе придется регулярно их обрабатывать.

Теперь рассмотрим проблему отсутствия: типовой безопасности в отношении операции восстановления значений из объектного образа. Вы знаете, что для восстановления значения в рамках синтаксиса C# используется оператор преобразования. Но каким будет это преобразование – успешным или неудачным, – выяснится только *в среде выполнения,*При попытке восстановить значение в неправильный тип данных вы получите InvalidCastException.

static void Main(string[] args) {

 …

**// Ой! Исключение времени выполнения!**

 Console.WriteLine("Значение вашего int: {0}", (short)myInts[0]);

 Console.ReadLine();

}

В идеальной ситуации компилятор C# должен решать проблемы некорректных операций восстановления из объектного образа во время компиляции, а не в среде выполнения. В связи с этим, в*действительно*идеальной ситуации, можно было бы сохранить типы, характеризуемые значениями, в контейнере, который не требовал бы преобразования в объект. В .NET 2.0 обобщения дают решение именно этих проблем. Однако перед тем как углубиться в детали использования обобщений, давайте посмотрим, как программисты пытались бороться с этими проблемами в .NET 1.x. с помощью строго типизованных коллекций.

**Типовая безопасность и строго типизованные коллекции**

В мире .NET, существовавшем до появления версии 2.0, программисты попытались решить проблемы типовой безопасности с помощью построения пользовательских строго типизованных коллекций. Для примера предположим, что вы хотите создать пользовательскую коллекцию, которая сможет содержать только объекты типа Person (персона).

public class Person {

**// Определены открытыми для простоты.**

 public int currAge;

 public string fName, lName;

 public Person(){}

 public Person(string firstName, string lastName, int age) {

  currAge = age;

  fName = firstName;

  lName = lastName;

 }

 public override string ToString() {

  return string.Format("Возраст {0}, {1} равен (2}", lName, fName, currAge);

 }

}

Чтобы построить коллекцию персон, можно определить член-переменную

System.Collections.ArrayList в рамках класса PeopleCollection и настроить все члены на работу со строго типизованными объектами Person, а не с общими объектами System.Object.

public class PeopleCollection: IEnumerable {

 private ArrayList arPeople = new ArrayList();

 public PeopleCollection(){}

**// Преобразование для вызывающей стороны.**

 public Person GetPerson(int pos) { return (Person)arPeople[pos]; }

**// Вставка только типов Person.**

 public void AddPerson(Person p) { arPeople.Add(p); }

 public void ClearPeople() { arPeople.Clear(); }

 public int Count { get { return arPeople.Count; } }

**// Поддержка foreach нумератора.**

 IEnumerator IEnumerable.GetEnumerator() { return arPeople.GetEnumerator(); }

}

С такими определениями типов вы будете уверены в типовой безопасности, поскольку теперь компилятор C# сможет распознать любую попытку вставки неподходящего типа,

static void Main (string[] args) {

 Console.WriteLine("\*\*\*\*\* Custom Person Collection *\*\*\*\*\*\*n");

 PeopleCollection myPeople = new PeopleCollection();

 myPeople.AddPerson(new Person("Homer", "Simpson", 40));

 myPeople.AddPerson(new Person("Marge", "Simpson", 38));

 myPeople.AddPerson(new Person("Lisa", "Simpson", 9));

 myPeople.AddPerson(new Person("Bart", "Simpson", 7));

 myPeople.AddPerson(new Person("Maggie", ''Simpson", 2));

**// Это приведет к ошибке компиляции!**

 myPeople.AddPerson(new Car());

 foreach (Person p in myPeople) Console.WriteLine(p);

 Console.ReadLine();

}

Хотя пользовательские коллекции и гарантируют типовую безопасность, при использовании этого подхода приходится создавать новые (и почти идентичные) пользовательские коллекции для каждого из соответствующих типов. Поэтому если вам понадобится пользовательская коллекция, которая должна работать только с классами, являющимися производными базового класса Car (автомобиль), вам придется построить очень похожий тип.

public class CarCollection: IEnumerable {

 private ArrayList arCars = new ArrayList();

 public CarCollection(){}

**// Преобразование для вызывающей стороны.**

 public Car GetCar(int pos) { return (Car) arCars[pos]; }

**// Вставка только типов Car.**

 public void AddCar(Car C) { arCars.Add(c); }

 public void ClearCars() { arCars.Clear(); }

 public int Count { get { return arCars.Count; } }

**// Поддержка foreach нумератора.**

 IEnumerator IEnumerable.GetEnumerator() { return arCars.GetEnumerator(); }

}

Вы, наверное, знаете из своего собственного опыта, что процесс создания множества строго типизованных коллекций для учета различных типов является не только трудоемким, но просто кошмарным для последующего обслуживания. Обобщенные коллекции позволяют отложить указание спецификации содержащегося типа до времени создания. Пока что не слишком беспокойтесь о синтаксических деталях. Рассмотрите следующий программный код, в котором используется обобщенный класс с именем System.Collections.Generic.List‹› для создания двух контейнерных объектов, обеспечивающих типовую безопасность.

static void Main(string [] args) {

**// Использование обобщенного типа List только для Person.**

 List‹Person› morePeople = new List‹Person›();

 morePeople.Add(new Person());

**// Использование обобщенного типа List только для Car.**

 List‹Car› moreCars = new List‹Car›();

**// Ошибка компиляции!**

 moreCars.Add(new Person());

}

**Проблемы создания объектных образов и строго типизованные коллекции**

Строго типизованные коллекции можно найти в библиотеках базовых классов .NET и это очень полезные программные конструкции. Однако эти пользовательские контейнеры мало помотают в решении проблем создания объектных образов. Даже если вы создадите пользовательскую коллекцию с именем IntCollection, предназначенную для работы только с типами данных System.Int32, вам придется создать объект некоторого типа для хранения самих данных (System.Array, System.Collections.ArrayList и т.п.).

public class IntCollection: IEnumerable {

 private ArrayList arInts = new ArrayList();

 public IntCollection() {}

**// Восстановление значения для вызывающей стороны.**

 public int GetInt(int pos) { return (int)arInts[pos]; }

**// Операция создания объектного образа!**

 public void AddInt(int i) { arInts.Add(i); }

 public void ClearInts() { arInts.Clear(); }

 public int Count { get { return arInts.Count; } }

 IEnumerator IEnumerable.GetEnumerator() { return arInts.GetEnumerator(); }

}

Вне зависимости от того, какой тип вы выберете для хранения целых чисел (System.Array, System.Collections.ArrayList и т.п.), вы не сможете избавиться от проблемы .NET 1.1, связанной с созданием объектных образов. Нетрудно догадаться, что здесь снова на помощь приходят обобщения. В следующем фрагменте программного кода тип System.Collections.Generic.List‹› используется для создания контейнера целых чисел, не имеющего проблем создания объектных образов и восстановлений значений при вставке и получении типов характеризуемых значений.

static void Main (string [] args) {

**// Баз создания объектного образа!**

 List‹int› myInts = new List‹int›();

 myInts.Add.(5);

**// Без восстановления значения!**

 int i = myInts[0];

}

Просто в качестве подтверждения рассмотрите следующий CIL-код для этого метода Main() (обратите внимание да отсутствие в нем каких бы то ни было блоков box и unbox).

.method private hidebysig static void Main(string[] args) cil managed {

 .entrypoint

**// Code size 24 (0x18)**

 .maxstack 2

 .locals init ([0] class [mscorlib] System.Collections.Generic.List`1‹int32› myInts, [1] int32 i)

 IL\_0000: nop

 IL\_0001: newobj instance void class [mscorlib] System.Collections.Generic.List`1‹int32›::.ctor()

 IL\_0006: stloc.0

 IL\_0007: ldloc.0

 IL\_0008: ldc.i4.5

 IL\_0009: callvirt instance void class [mscorlib]System.Collections.Generic.List`1‹int32›::Add(!0)

 IL\_000e: nop

 IL\_000f: ldloc.0

 IL\_0010: ldc.i4.0

 IL\_0011: callvirt instance !0 class [mscorlib]System.Collections.Generic.List`1‹int32›::get\_Item(int32)

 IL\_0016: stloc.1

 IL\_0017: ret

} // end of method Program::Main

Теперь, когда вы имеете лучшее представление о роли обобщений в .NET2.0, мы с вами готовы углубиться в детали. Для начала мы формально рассмотрим пространство имен System.Collections.Generic.

**Исходный код.**Проект CustomNonGenericCollection размещен в подкаталоге, соответствующем главе 10.

**Пространство имен System.Collections.Generic**

Обобщенные типы присутствуют во многих библиотеках базовых классов .NET 2.0, но пространство имен System.Collections.Generic буквально наполнено ими (что вполне соответствует его названию). Подобно своему "родственнику" без обобщений (System.Collections), пространство имен System.Collections. Generic содержит множество типов класса и интерфейса, что позволяет вкладывать элементы в самые разные контейнеры. Совсем не удивительно, что обобщенные интерфейсы имитируют соответствующие необобщенные типы из пространства имен System.Collections.

• ICollection‹T›

• IComparer‹T›

• IDictionary‹K, V›

• IEnumerable‹T›

• IEnumerator‹T›

• IList‹T›

**Замечание.**По соглашению для обобщенных типов их замещаемые параметры обозначаются буквами верхнего регистра. И хотя здесь допустимо использовать любые буквы (или слова), обычно используют Т для обозначения типов, К – для ключей, а V – для значений.

В пространстве имен System.Collections.Generic определяется и ряд классов, реализующих многие из этих ключевых интерфейсов. В табл. 10.1 представлены описания базовых типов класса из этого пространства имен, реализуемые ими интерфейсы и соответствующие типы из пространства имен System.Collections.

В пространстве имен System.Collections.Generic также определяется целый ряд "вспомогательных" классов и структур для работы с конкретными контейнерами. Например, тип LinkedListNode‹T› представляет узел в обобщенном LinkedList‹T›, исключение KeyNotFoundException возникает при попытке доступа к элементу контейнера с несуществующим ключом и т.д.

Как видно из табл. 10.1, многие обобщенные классы коллекции имеют необобщенные аналоги в пространстве имен System.Collections (иногда даже с одинаковыми именами). В главе 7 было показано, как работать с такими необобщенными типами, поэтому дальше не предполагается рассматривать все их обобщенные "дубликаты". Мы рассмотрим только List‹T›, чтобы проиллюстрировать приемы использования обобщений. Если вам нужны подробности о других элементах пространства имен System.Collections.Generic, обратитесь к документации .NET Framework 2.0.

**Таблица 10.1.**Классы System.Collections.Generic

|  |  |  |
| --- | --- | --- |
| **Обобщенный класс** | **Необобщенный аналог в System.Collections** | **Описание** |
| Collection‹T› | CollectionBase | База для обобщенной коллекции |
| Comparer‹T› | Comparer | Выполняет сравнение двух обобщенных объектов |
| Dictionary‹K, V› | Hashtable | Обобщенная коллекция пар имен и значений |
| List‹T› | ArrayList | Список элементов с динамически изменяемыми размерами |
| Queue‹T› | Queue | Обобщенная реализация списка FIFO (дисциплина обслуживания типа "очередь") |
| SortedDictionary‹K, V› | SortedList | Обобщенная реализаций сортированного набора пар имен и значений |
| Stack<T> | Stack | Обобщенная реализация списка LIFO (дисциплина обслуживания типа "стек") |
| LinkedList‹T› | - | Обобщенная реализация двусвязного списка |
| ReadOnlyCoIlection‹T› | ReadOnlyCollectionBase | Обобщенная реализация набора элементов только для чтения |

**Тип List‹T›**

Подобно необобщенным классам, обобщенные классы являются объектами, размещаемыми в динамической памяти, поэтому для них следует использовать new со всеми необходимыми аргументами конструктора. Кроме того, вы должны указать типы, замещающие параметры, определенные обобщенным типом. Так, для System.Collections.Generic.List‹T› требуется указать одно значение, задающее вид элемента, с которым будет функционировать List‹T›*.*Например, чтобы создать три объекта List‹› для хранения целых чисел, объектов SportsCar и объектов Person, вы должны записать следующее

static void Main(string[] args) {

**// Создается List для хранения целых чисел.**

 List‹int› myInts = new List‹int›();

**// Создается List для хранения объектов SportsCar.**

 List‹SportsCar› myCars = new List*‹*SportsCar›();

**// Создается List для хранения объектов Person.**

 List‹Person› myPeople = new List‹Person›();

}

В этот момент вы можете поинтересоваться, что же на самом деле становится значением заполнителя. Открыв окно определения программного кода в Visual Studio 2005 (см. главу 2), вы увидите, что везде в определении типа List‹T› используется заполнитель Т. Ниже показана часть соответствующего листинга (обратите внимание на элементы, выделенные полужирным шрифтом).

**// Часть листинга для типа List‹T›.**

namespace System.Collections.Generic {

 public class List‹**T**›: IList‹**T**›, ICollection‹**T**›, IEnumerable‹**T**›, IList, ICollection, IEnumerable {

  …

  public void Add(**T** item);

  public IList‹**T**› AsReadOnly();

  public int BinarySearch(**T** item);

  public bool Contains(**T** item)*;*

  public void CopyTo(**T**[] array);

  public int FindIndex(System.Predicate‹**T**› match);

  public T FindLast(System.Predicate‹**T**› match);

  public bool Remove(**T** item);

  public int RemoveAll(System.Predicate‹**T**› match);

  public **T**[] ToArray();

  public bool TrueForAll(System.Predicate‹**T**› match);

  public **T** this[int index] { get; set; }

  …

 }

}

Когда вы создаете тип List‹T› и указываете для него SportsCar, это эквивалентно следующему определению типа List‹T›.

namespace System.Collections.Generic {

 public class List‹**SportsCar**›: IList‹**SportsCar**›, ICollection‹**SportsCar**›, IEnumerable‹**SportsCar**›, IList, ICollection, IEnumerable {

  …

  public void Add(**SportsCar** item);

  public IList‹**SportsCar**› AsReadOnly()*;*

  public int BinarySearch(**SportsCar** item);

  public bool Contains(**SportsCar** item);

  public void CopyTo(**SportsCar**[] array);

  public int FindIndex(System.Predicate‹**SportsCar**› match);

  public SportsCar FindLast(System.Predicate‹**SportsCar**› match);

  public bool Remove(**SportsCar** item);

  public int RemoveAll(System.Predicate‹**SportsCar**› match);

  publiс **SportsCar**[] ToArray();

  public bool TrueForAll(System.Predicate‹**SportsCar**› match);

  public **SportsCar** this[int index] { get; set; }

  …

 }

}

Конечно, когда вы создаете обобщенный List‹T›, нельзя сказать, что компилятор буквально создает совершенно новую реализацию типа List‹T›. Он обращается только к тем членам обобщенного типа, которые вы вызываете фактически. Чтобы пояснить это, предположим, что вы используете List‹T› для объектов SportsCar так.

static void Main(string[] args) {

**// Проверка List, содержащего объекты SportsCars.**

 List‹SportsCar› myCars = new List‹SportsCar›();

 myCars.Add(new SportsCar());

 Console.WriteLine("Your List contains {0}", myCars.Count);

}

Если с помощью ildasm.exe проверить генерируемый CIL-код, обнаружатся следующие подстановки.

.method private hidebysig static void Main(string[] args) cil managed {

 .entrypoint

 .maxstack 2

 .locals init ([0] class [mscorlib] System.Collections.Generic.'List`1'‹class **SportsCar›**myCars)

 newobj instance void class [mscorlib]System.Collections.Generic.'List`1'‹class **SportsCar›**::.ctor()

 stloc.0

 ldloc.0

 newobj instance void CollectionGenerics.SportsCar::.ctor()

 callvirt instance void class [mscorlib]System.Collections.Generic.'List`1'‹class **SportsCar›::**Add(!0)

 nop

 ldstr "Your List contains {0} item(s)."

 ldloc.0

 callvirt instance int32 class [mscorlib] System.Collections.Generic.'List`1' ‹class **SportsCar**›::get\_Count()

 box [mscorlib] System.Int32

 call void [mscorlib]System.Console::WriteLine(string, object)

 nop

 ret

}

Теперь, после изучения процесса использования обобщенных типов из библиотек базовых классов, в оставшейся части главы мы рассмотрим вопросы создания обобщенных методов, типов и коллекций.

**Создание обобщенных методов**

Чтобы научиться интегрировать обобщения в проекты, мы начнем с простого примера обычной подпрограммы свопинга. Целью этого примера является построение метода обмена, который сможет работать c любыми типами данных (характеризуемыми значениями или ссылками), используя для этого один параметр типа. В силу самой природы алгоритмов свопинга входные параметры будут посылаться по ссылке (с помощью ключевого слова C# ref). Вот соответствующая полная реализация.

**// Этот метод переставляет любые два элемента,**

**// определенные параметром типа ‹Т›.**

static void Swap‹T›(ref T a, ref Т b) {

 Console.WriteLine ("Методу Swap () передано {0}", typeof(T));

 Т temp;

 temp = а;

 а = b;

 b = temp;

}

Обратите внимание на то, что обобщенный метод определяется с помощью указания параметра типа, размещаемого после имени метода, но перед списком параметров. Здесь вы заявляете, что метод Swap() может работать с любыми двумя параметрами типа ‹Т›. Просто для информации вы выводите имя типа соответствующего заменителя на консоль с помощью оператора C# typeof(). Теперь рассмотрите следующий метод Main(), в котором происходит обмен между целочисленными и строковыми типами.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с обобщениями \*\*\*\*\*\n")*;*

**// Обмен между двумя целыми.**

 int а = 10, b = 90;

 Console.WriteLine("До обмена: {0}, {l}", а, b);

 Swap‹int›(ref a, ref b);

 Console.WriteLine("После обмена: {0}, {1}", а, b);

 Console.WriteLine();

**// Обмен между двумя строками.**

 string s1 = "Hello", s2 = "There";

 Console.WriteLine("До обмена: {0} {1}!", s1, s2);

 Swap‹string›(ref s1, ref s2);

 Console.WriteLine("После обмена: {0} {1}!", s1, s2);

 Console.ReadLine();

}

**Пропуск параметров типа**

При вызове обобщенных методов, подобных Swap‹T›, у ваc есть возможность не указывать параметр типа, но только в том случае, когда обобщенный метод требует указания аргументов, поскольку тогда компилятор может "выяснить" тип этих аргументов на основе вводимых параметров. Например, можно переставить два типа System.Boolean так.

**// Компилятор будет предполагать System.Boolean.**

bool b1 = true, b2 = false;

Console.WriteLine("До обмена: {0}, {1}", b1, b2);

**Swap(ref b1**, **ref b2)**;

Console.WriteLine("После обмена: {0}, {1}", b1, b2);

Но если, например, у вас есть обобщённый метод с именем DisplayBaseClass‹T›, не имеющий входных параметров, как показано ниже:

static void DisplayBaseClass‹T›() {

 Console.WriteLine("Базовым классом {0} является: {1}.",  typeof(T), typeof(Т).BaseType);

}

то при вызове этого метода вы должны указать параметр типа.

static void Main(string[] args) {

…

**// Если метод не имеет параметров,**

**// необходимо указать параметр типа.**

 DisplayBaseClass‹int›();

 DisplayBaseClass‹string›();

**// Ошибка компиляции!**

**// Нет параметров? Тогда должен быть заполнитель!**

 DisplayBaseClass();

 …

}



Рис. 10.1. Обобщенные методы в действии

В данном случае обобщенные методы Swap‹T› и DisplayBaseClass‹T› были определены в рамках объекта приложения (т.е. в рамках типа, определяющего метод Main()). Если вы предпочтете определить эти члены в новом типе класса (MyHelperClass), то должны записать следующее.

public class MyHelperClass {

 public static void Swap‹T›(ref T a, ref T b) {

  Console.WriteLine("Методу Swap() передано {0}", typeof(T));

  T temp;

  temp = a;

  a = b;

  b = temp;

 }

 public static void DisplayBaseClass‹T›() {

  Console.WriteLine("Базовым классом {0} является: {1}.", typeof(T), typeof(T).BaseType);

 }

}

Обратите внимание на то, что тип MyHelperClass сам по себе не является обобщенным, но определяет два обобщенных метода. Так или иначе, теперь, когда методы Swap‹T› и DisplayBaseClass‹T› находятся в контексте нового типа класса, при вызове их членов придется указать имя типа.

MyHelperClass.Swap‹int›(ref a, ref b);

Наконец, обобщенные методы не обязаны быть статическими. Если бы Swap‹T› и DisplayBaseClass‹T› были методами уровня экземпляра, нужно было бы просто создать экземпляр MyHelperClass и вызвать их из объектной переменной.

MyHelperClass с = new MyHelperClass();

c.Swap‹int›(ref a, ref b);

**Создание обобщенных структур (и классов)**

Теперь, когда вы понимаете, как определять и вызывать обобщенные методы, давайте рассмотрим построение обобщенных структур (процедура построения обобщенных классов оказывается аналогичной). Предположим, что мы построили гибкую структуру Point, поддерживающую один параметр типа, который представляет единицу хранения координат (х, у)*.*Вызывающая сторона может создавать типы Point‹T› так.

// **Point с использованием int.**

Point‹int› p = new Point‹int›(10, 10);

// **Point с использованием double.**

Point‹double› p2 = new Point‹double›(5.4, 3.3);

Вот полное определение Point‹T›, необходимое нам для дальнейшего анализа.

**// Обобщенная структура Point.**

public struct Point‹T› {

**// Обобщенные данные**

 private T xPos;

 private T yPos;

**// Обобщенный конструктор.**

 public Point (T xVal, T yVal) {

  xPos = xVal;

  yPos = yVal;

 }

**// Обобщенные свойства.**

 public T X {

  get {return xPos;}

  set {xPos = value;}

 }

 public T Y {

  get { return yPos; }

  set { yPos = value; }

 }

 public override string ToString() {

  return string.Format("[{0}, {1}]", xPos*,*yPos);

 }

**// Переустановка полей со значениями параметра типа,**

**// принятыми по умолчанию.**

 public void ResetPoint() {

  xPos = default(T);

  yPos = default(T);

 }

}

**Ключевое слово default в обобщенном программном коде**

Как ведите, Point‹T› использует параметр типа в определении полей данных, аргументов конструктора и в определениях свойств. Обратите внимание на то, что вдобавок к переопределению ToString() обобщенный тип Point‹T› определяет метод ResetPoint(), в котором используется новый синтаксис.

**// Ключевое слово 'default' в C# 2005 является перегруженным.**

**// При использовании с обобщениями оно представляет значение**

**// параметра типа, принимаемое по умолчанию.**

public void ResetPoint() {

 xPos = default(Т);

 yPos = default(T);

}

В C# 2005 ключевое слово default получило два значения. Кроме использования в конструкции switch, оно может использоваться для установки параметрам типа значений, принятых по умолчанию. И это, очевидно, полезно, поскольку обобщенный тип ничего заранее не знает о фактических замещающих значениях и поэтому не может с безопасностью предполагать о том. каким должно быть значение по умолчанию. Значения по умолчанию для параметра типа являются следующими.

• Для числовых значений значением по умолчанию является 0.

• Для ссылочных типов значением по умолчанию является null.

• Поля структуры устанавливаются равными 0 (для типов, характеризуемых значениями) или null (для ссылочных типов).

Для Point‹T› вы можете непосредственно установить xPos и yPos равными 0, поскольку вполне безопасно предполагать, что вызывающая сторона будет поставлять только числовые данные. Однако с помощью синтаксиса default(T) вы можете сделать обобщенный тип более гибким. В любом случае вы теперь можете использовать методы Point‹T› так.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с обобщениями \*\*\*\*\*\n");

**// Point с использованием int.**

 Point‹int› p = new Point‹int›(10, 10);

 Console.WriteLine("p.ToString()={0}", p.ToString());

 p.ResetPoint();

 Console.WriteLine("p.ToString()={0}", p.ToString());

 Console.WriteLine();

**// Point с использованием double.**

 Point‹double› p2 = new Point‹double›(5.4, 3.3);

 Console.WriteLine("p2.ToString()={0}", p2.ToString());

 p2.ResetPoint();

 Console.WriteLine("p2.ToString()={0}", p2.ToString());

 Console.WriteLine();

**// Обмен двух Point.**

 Point‹int› pointA = new Point‹int›(50, 40);

 Point‹int› pointB = new Point‹int›(543, 1);

 Console.WriteLine("До обмена: {0}, {1}", pointA, pointB);

 Swap‹Point‹int> >(ref pointA, ref pointB);

 Console.WriteLine("После обмена: {0}, {1}", pointA, pointB);

 Console.ReadLine();

}

Соответствующий вывод показан на рис. 10.2.

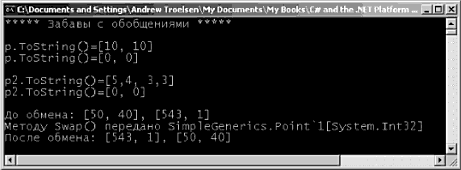


Рис. 10.2. Использование обобщённого типа Point

**Исходный код**. Проект SimpleGenerics размещен в подкаталоге, соответствующем главе 10.

**Создание пользовательских обобщенных коллекций**

Итак, пространство имен System.Collections.Generic предлагает множество типов, позволяющих создавать эффективные контейнеры, удовлетворяющие требованиям типовой безопасности. С учетом множества доступных вариантов очень велика вероятность того, что в .NET 2.0 у вас вообще не возникнет необходимости в построении пользовательских типов коллекции. Тем не менее, чтобы показать, как строится обобщенный контейнер, нашей следующей задачей будет создание обобщенного класса коллекции, который мы назовем CarCollection‹Т›.

Подобно созданному выше необобщенному типу CarCollection, наш новый вариант будет использовать уже существующий тип коллекции для хранения своих элементов (в данном случае это List‹›). Будет реализована и поддержка цикла foreach путем реализации обобщенного интерфейса IEnumerable‹›. Обратите внимание на то, что IEnumerable‹› расширяет необобщенный интерфейс IEnumerable, поэтому компилятор ожидает, что вы реализуете *две*версии метода GetEnumerator(). Вот как может выглядеть соответствующая модификация.

public class CarCollection‹T›: IEnumerable‹T› {

 private List‹T› arCars = new List‹T›();

 public T GetCar(int pos) { return arCars[pos]; }

 public void AddCar(T c) { arCars.Add(c); }

 public void ClearCars() { arCars.Clear(); }

 public int Count { get { return arCars.Count; } }

**// IEnumerable‹T› расширяет IEnumerable, поэтому**

**// нужно реализовать обе версии GetEnumerator()**.

 IEnumerator‹T› IEnumerable‹Т›.GetEnumerator() { return arCars.GetEnumerator(); }

 IEnumerator IEnumerable.GetEnumerator() { return arCars.GetEnumerator(); }

}

Этот обновленный тип CarCollection‹T› можно использовать так.

static void Main(string[] args) {

 Console.WriteLine("\* Пользовательская обобщенная коллекция \*\n");

**// Создание коллекции объектов Car.**

 CarCollection‹Car› myCars = new CarColleetion‹Car›();

 myCars.AddCar(new Car("Rusty", 20));

 myCars.AddCar(new Car("Zippy", 90));

 foreach(Car c in myCars) {

  Console.WriteLine("PetName: {0}, Speed: {1}", с.PetName, с.Speed);

 }

 Console.ReadLine();

}

Здесь создается тип CarCollection‹T›, который должен содержать только типы Car. Снова заметим, что того же результата можно достичь и с помощью непосредственного использования типа List‹T›. Плавным преимуществом данного подхода является то, что теперь вы можете добавлять в CarCollection уникальные методы, делегирующие запросы к внутреннему типу List‹T›.

**Установка ограничений для параметров типа с помощью where**

В настоящий момент класс CarCollection‹T› привлекает нас только открытыми методами с уникальными именами. Кроме того, пользователь объекта может создать экземпляр CarCollection‹T› и указать практически любой параметр типа.

**// Это синтаксически корректно, но выглядит,**

**// по крайней мере, странно…**

CarCollection‹int› myInts = new CarCollection‹int›();

myInts.AddCar(5);

myInts.AddCar(11);

Чтобы проиллюстрировать другую форму типичного непредусмотренного использования объекта, предположим, что вы создали два новых класса – SportsCar (спортивная машина) и MiniVan (минивэн), – которые являются производными от Car.

public class SportsCar: Car {

 public SportsCar(string p, int s): base(p, s){}

**// Дополнительные методы для SportsCar.**

}

public class MiniVan: Car {

 public MiniVan(string p, int a): base(p, s) {}

**// Дополнительные методы для MiniVan.**

}

В соответствии с законами наследования, в коллекцию CarCollection‹T›, созданную с параметром типа Car, можно добавлять и типы MiniVan и SportsCar.

**// CarCollection‹Car› может хранить любой тип, производный от Car.**

CarCollection‹Car› myCars = new CarCollection‹Car›();

myInts.AddCar(new MiniVan("Family Truckster", 55);

myInts.AddCar(new SportsCar("Crusher", 40));

Это синтаксически корректно, но что делать, если вдруг понадобится добавить в CarCollection‹T› новый открытый метод, например, с именем PrintPetName()? Такая задача кажется простой – достаточно получить доступ к подходящему элементу из List‹T› и вызвать свойство PetName.

**// Ошибка!**

**// System.Объект не имеет свойства о именем PetName.**

public void PrintPetName(int pos) {

 Console.WriteLine(arCars[pos].PetName);

}

Однако в таком виде программный код скомпилирован не будет, поскольку истинная суть ‹Т› еще не известна, и вы не можете с уверенностью утверждать, что какой-то элемент типа List‹T› будет иметь свойство PetName. Когда параметр типа не имеет никаких ограничений (как в данном случае), обобщенный тип называется *свободным*(unbound). По идее параметры свободного типа должны иметь только члены System.Object (которые, очевидно, не имеют свойства PetName).

Вы можете попытаться "обмануть" компилятор путем преобразования элемента, возвращенного из Метода индексатора List‹T›, в строго типизованный объект Car, чтобы затем вызвать petName возвращенного объекта.

**// Ошибка!**

**// Нельзя превратить тип 'Т' в 'Car'!**

public void PrintPetName(int pos) {

 Console.WriteLine(((Car)arCars[pos]).PetName);

}

Но это тоже не компилируется, поскольку компилятор не знает значения параметра типа ‹Т› и не может гарантировать, что преобразование будет законным.

Для решения именно таких проблем обобщения .NET могут опционально определяться с ограничениями, для чего используется ключевое слово where. В .NET 2.0 обобщения могут иметь ограничения, описанные в табл. 10.2.

Таблица 10.2. Возможные ограничения обобщений для параметров типа

|  |  |
| --- | --- |
| **Ограничение обобщения** | **Описание** |
| where T: struct | Параметр типа ‹T› должен иметь в цепочке наследования System.ValueType |
| where T: class | Параметр типа ‹T› *не*должен иметь в цепочке наследования System.ValueType (т.е. ‹Т› должен быть ссылочным типом) |
| where T: new() | Параметр типа ‹T› должен иметь конструктор, заданный по умолчанию. Это полезно тогда, когда обобщенный тип должен создать экземпляр параметра типа, а вы не имеете ясных предположений о формате пользовательских конструкторов. Заметьте, что это ограничение должно быть последним в списке ограничений, если у типа их несколько |
| where T:*БазовыйКласс* | Параметр типа ‹T› должен быть производным класса, указанного параметром *БазовыйКласс* |
| where T:*Интерфейс* | Параметр типа ‹T› должен реализовывать интерфейс, указанный параметром *Интерфейс* |

При наложении ограничений с помощью ключевого слова where список ограничений размещается после имени базового класса обобщенного типа и списка интерфейсов. В качестве конкретных примеров рассмотрите следующие ограничения обобщенного класса MyGenericClass.

**// Вложенные элементы должны иметь конструктор,**

**// заданный по умолчанию.**

public class MyGenericClass‹T› where T: new() {…}

**// Вложенные элементы должны быть классами, реализующими IDrawable**

**// и поддерживающими конструктор, заданный по умолчанию.**

public class MyGenericClass‹T› where T: class, IDrawable, new() {…}

**// MyGenericClass получается из МуВаsе и реализует ISomeInterface,**

**// а вложенные элементы должны быть структурами.**

public class MyGenericClass‹T›: MyBase, ISomeInterface where T: struct {…}

При построении обобщенного типа, в котором указано несколько параметров типа, вы можете указать уникальный набор ограничений для каждого из таких параметров.

**// ‹К› должен иметь конструктор, заданный по умолчанию,**

**// а ‹Т› должен реализовывать открытый интерфейс IComparable.**

public class MyGenericClass‹K, T› where K: new() where T: IComparable‹T› {…}

Если вы хотите изменить тип CarCollection‹T› так, чтобы в него можно было поместить только производные от Car, вы можете записать следующее.

public' class CarCollection‹T›: IEnumerable‹T› where T: Car {

 public void PrintPetName(int роs) {

**// Поскольку теперь все элементы должны быть из семейства Car,**

**// свойство PetName можно вызывать непосредственно**.

  Console.WriteLine(arCars[pos].PetName);

 }

}

При таких ограничениях на CarCollection‹T› реализация PrintPetName() становится очень простой, поскольку теперь компилятор может предполагать, что ‹Т› является производным от Car. Более того, если указанный пользователем параметр типа не совместим с Car, будет сгенерирована ошибка компиляции.

**// Ошибка компиляции!**

CarCollection‹int› myInts = new CarCollection‹int›();

Вы должны понимать, что обобщенные методы тоже могут использовать ключевое слово where. Например, если нужно гарантировать, чтобы методу Swap(), созданному в этой главе выше, передавались только типы, производные от System. ValueType, измените свой программный код так.

**// Этот метод переставит любые типы, характеризуемые значениями.**

static void Swap‹T›(ref Т а, ref T b) where T: struct {

 …

}

Следует также понимать то, что при таком ограничении метод Swap() уже не сможет переставлять строковые типы (поскольку они являются ссылочными).

**Отсутствие поддержки ограничений при использовании операций**

При создании обобщенных методов для вас может оказаться сюрпризом появление ошибок компилятора, когда с параметрами типа используются операции C# (+, -, \*, == и т.д.). Например, я уверен, вы сочли бы полезными классы Add(), Subtract(), Multiply() и Divide(), способные работать с обобщенными типами.

**// Ошибка компиляции!**

**// Нельзя применять операции к параметрам типа!**

public class BasicMath‹T› {

 public T Add(T arg1, T arg2) { return arg1 + arg2; }

 public T Subtract(T arg1, T arg2) { return arg1 – arg2; }

 public T Multiply(T arg1, T arg2) { return arg1 \* arg2; }

 public T Divide(T arg1, T arg2) { return arg1 / arg2; }

}

Как ни печально, этот класс BasicMath‹T› не компилируется. Это может показаться большим ограничением, но не следует забывать, что обобщения являются *обобщениями.*Конечно, тип System.Int32 может прекрасно работать с бинарными операциями C#. Однако, если, например, ‹T› будет пользовательским классом иди типом структуры, компилятор не сможет сделать никаких предположений о характере перегруженных операций +, -, \* и /. В идеале C# должен был бы позволять обобщенному типу ограничения с использованием операций, например, так.

**// Только для иллюстрации!**

**// Этот программный код не является допустимым в C# 2.0.**

public class BasicMath‹T› where T: operator +, operator -, operator \*, operator / {

 public T Add(T arg1, T arg2) { return arg1 + arg2; }

 public T Subtract(T arg1, T arg2) { return arg1 – arg2; }

 public T Multiply(T arg1, T arg2) { return arg1 \* arg2; }

  public T Divide(T arg1, T arg2) { return arg1 / arg2; }

}

Увы, ограничения обобщенных типов при использовании операций в C# 2005 не поддерживаются.

**Исходный код.**Проект CustomGenericCollection размещен в подкаталоге, соответствующем главе 10.

**Создание обобщенных базовых классов**

Перед рассмотрением обобщенных интерфейсов следует указать на то, что обобщенные классы могут быть базовыми для других классов и могут таким образом определять любое число виртуальных и абстрактных методов. Однако производные типы должны подчиняться определенным правилам, вытекающим из природы обобщенной абстракции. Во-первых, если обобщенный класс расширяется необобщенным, то производный класс должен конкретизировать параметр типа,

**// Предположим, что создан пользовательский**

**// обобщенный класс списк**а.

public class MyList‹T› {

 private List‹T› listOfData = new List‹T›();

}

**// Конкретные типы должны указать параметр типа,**

**// если они получаются из обобщенного базового класса.**

public class MyStringList: MyList‹string› {}

Кроме того, если обобщенный базовый класс определяет обобщенные виртуальные или абстрактные методы, производный тип должен переопределить эти обобщенные методы, используя конкретизированный параметр типа.

**// Обобщенный класс с виртуальным методом.**

public class MyList‹T› {

 private List‹T› listOfData = new List‹T›();

 public virtual void PrintList(T data) {}

}

public class MyStringList: MyList‹string› {

**// В производных методах нужно заменить параметр типа,**

**// используемый а родительском классе.**

 public override void PrintList(string data) {}

}

Если производный тип тоже является обобщенным, дочерний класс может (опционально) использовать заменитель типа в своем определении. Однако знайте, что любые ограничения, размещенные в базовом классе, должны "учитываться" и производным типом. Например:

**// Обратите внимание, теперь здесь имеется ограничение,**

**// требующее конструктор по умолчанию.**

public class MyList‹T› where T: new() {

 private List‹T› listOfData = new List‹T›();

 public virtual void PrintList(T data) {}

**// Производный тип должен учитывать ограничения базового.**

 public class MyReadOnlyList‹T›: MyList‹T› where T: new() {

  public override void PrintList(T data) {}

}

Если вы только не планируете построить свою собственную библиотеку обобщений*,*вероятность того, что вам придется строить иерархии обобщенных классов, оказывается практически нулевой. Тем не менее, вы теперь знаете, что язык C# обеспечивает поддержку наследования обобщений.

**Создание обобщенных интерфейсов**

Вы уже видели при рассмотрении пространства имен System.Collections. Generiс, что обобщенные интерфейсы в C# также допустимы (например, IEnumerable‹Т›). Вы, конечно, можете определить свои собственные обобщенные интерфейсы (как с ограничениями, так и без ограничений). Предположим, что нужно определить интерфейс, который сможет выполнять бинарные операции с параметрами обобщенного типа.

public interface **IBinaryOperations‹T›**{

**T** Add(**T** arg1, **T** arg2);

**T** Subtract(**T** arg1, **T** arg2);

**T** Multiply(**T** arg1, **T** arg2);

**T**Divide(**T** arg1, **T** arg2);

}

Известно, что интерфейсы остаются почти бесполезными, пока они не реализованы некоторым классом или структурой. При реализации обобщенного интерфейса поддерживающий его тип указывает тип заполнителя.

public class BasicMath: **IBinaryOperations‹int›**{

 public **int**Add(**int** arg1, **int** arg2) { return arg1 + arg2; }

 public **int** Subtract(**int** arg1, **int** arg2) { return arg1 – arg2; }

 public **int**Multiply(**int** arg1, **int** arg2) { return arg1 \* arg2; }

 public **int** Divide(**int** arg1, **int** arg2) { return arg1 / arg2; }

}

После этого вы можете использовать BasicMath, как и ожидали.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Обобщенные интерфейсы \*\*\*\*\*\n");

 BasicMath m = new BasicMath();

 Console.WriteLine("1 + 1 = {0}", m.Add(1, 1));

 Console.ReadLine();

}

Если вместо этого требуется создать класс BasicMath, действующий на числа с плавающим десятичным разделителем, можно конкретизировать параметр типа так.

public class BasicMath: IBinaryOperations‹double› {

 public **double**Add(**double**arg1, **double**arg2) { return arg1 + arg2; }

 …

}

**Исходный код.**Проект GenericInterface размещен в подкаталоге, соответствующем главе 10.

**Создание обобщенных делегатов**

Наконец, что не менее важно, .NET 2.0 позволяет определять обобщенные типы делегата. Предположим, например, что требуется определить делегат, который сможет вызывать любой метод, возвращающий void и принимающий один аргумент. Если аргумент может меняться, это можно учесть с помощью параметра типа. Для примера рассмотрим следующий программный код (обратите внимание на то, что целевые объекты делегата регистрируются как с помощью "традиционного" синтаксиса делегата, так и с помощью группового преобразования метода).

namespace GenericDelegate {

**// Этот обобщенный делегат может вызвать любой метод,**

**// возвращающий void и принимающий один параметр.**

 public delegate void MyGenericDelegate‹T›(T arg);

 class Program {

  static void Main(string[] args) {

   Console.WriteLine("\*\*\*\*\* Обобщенные делегаты \*\*\*\*\*\n");

**// Регистрация цели с помощью 'традиционного'**

**// синтаксиса делегата.**

   MyGenericDelegate‹string› strTarget = new MyGenericDelegate‹string›(StringTarget);

   strTarget("Некоторые строковые данные");

**// Регистрация цели с помощью**

**// группового преобразования метода.**

   MyGenericDelegate‹int› intTarget = IntTarget;

   intTarget(9);

   Console.ReadLine();

  }

  static void StringTarget(string arg) {

   Console.WriteLine("arg в верхнем регистре: {0}", arg.ToUpper());

  }

  static void IntTarget(int arg) {

   Console.WriteLine("++arg: {0}", ++arg);

  }

 }

}

Обратите внимание на то. что MyGenericDelegate‹T› определяет один пара-метр типа, представляющий аргумент, отправляемый целевому объекту делегата. При создании экземпляра этого типа требуется конкретизировать значение параметра типа, а также имя метода, вызываемого делегатом. Так, если вы укажете строковый тип, то отправите целевому методу строковое значение.

**// Создание экземпляра MyGenericDelegate‹T›**

**// со значением string для параметра типа.**

MyGenericDelegate‹string› strTarget = new MyGenericDelegate‹string›(StringTarget);

strTarget("Некоторые строковые данные");

С учетом формата объекта strTarget метод StringTarget() должен теперь получить в качестве параметра одну строку.

static void StringTarget(string arg) {

 Console.WriteLine("arg в верхнем регистре: {0}", arg.ToUpper());

}

**Имитация обобщенных делегатов в .NET 1.1**

Как видите, обобщенные делегаты предлагают более гибкий подход для указания вызываемых методов. В рамках .NET 1.1 аналогичного результата можно достичь с помощью базового System.Object.

public delegate void MyDelegate(object arg);

Это позволяет посылать любой тип данных целевому объекту делегата, но при этом не гарантируется типовая безопасность, а кроме того, остаются нерешенными проблемы создания объектных образов. Предположим, например, что мы создали два экземпляра MyDelegate, которые указывают на один и тот же метод MyTarget. Обратите внимание на проблемы создания объектных образов и восстановления значений, а также на отсутствие гарантий типовой безопасности.

class Program {

 static void Main(string[] args) {

  …

**// Регистрация цели с помощью**

**// 'традиционного' синтаксиса делегата.**

  MyDelegate d = new MyDelegate(MyTarget) d("Дополнительные строковые данные");

**// Регистрация цели с помощью**

**// группового преобразования метода.**

  MyDelegate d2 = MyTarget;

  d2(9);**// Проблема объектного образа.**

  …

 }

**// Ввиду отсутствия типовой безопасности мы должны**

**// определить соответствующий тип до преобразования.**

 static void MyTarget(object arg) {

  if (arg is int) {

   int i = (int)arg;**// Проблема восстановления значения.**

   Console.WriteLine("++arg: {0}", ++i);

  }

  if (arg is string) {

   string s *=*(string) arg;

   Console.WriteLine("arg в верхнем регистре: {0}", s.ToUpper());

  }

 }

}

Когда вы посылаете целевому объекту тип, характеризуемый значением, это значение (конечно же) "упаковывается" в объект и "распаковывается" после его получения целевым методом. Точно так же, поскольку поступающий параметр может быть всем чем угодно, вы должны динамически проверить соответствующий тип перед тем, как выполнить преобразование. С помощью обобщенных делегатов вы можете получить всю необходимую гибкость без "проблем".

**Несколько слов о вложенных делегатах**

Завершим эту главу рассмотрением еще одного аспекта обобщенных делегатов. Вы знаете, что делегаты могут быть вложены в тип класса, что должно означать тесную ассоциацию между этими двумя ссылочными типами. Если тип-контейнер при этом оказывается обобщенным, вложенный делегат может использовать в своем определении параметры типа.

**// Вложенные обобщающе делегаты могут иметь доступ к параметрам**

**// обобщенного типа-контейнера.**

public class MyList‹T› {

 private List‹T› listOfData = new List‹T›();

 public delegate void ListDelegate(T arg);

}

**Исходный код**. Проект GenericDetegate размещен в подкаталоге, соответствующем главе 10.

**Резюме**

Обобщения можно обоснованно считать главным из усовершенствований, предложенных в C# 2005. Как вы могли убедиться, обобщенный элемент позволяет указать "заполнители" (т.е. параметры типа), которые конкретизируются в момент создания типа (или вызова, в случае обобщенных методов). По сути, обобщения дают решение проблем объектных образов и типовой безопасности, усложняющих разработку программ в среде .NET 1.1.

Чаще всего вы будете просто использовать обобщенные типы, предлагаемые библиотеками базовых классов .NET, но можно создавать и свои собственные обобщенные типы. При создании обобщенных типов можно указать любое число ограничений, чтобы повысить уровень типовой безопасности и гарантировать, что соответствующие операции будут выполняться с "известными величинами".

**ЧАСТЬ III. Программирование компоновочных блоков .NET**

**ГЛАВА 11. Компоновочные блоки .NET**

Любое из приложений, описанных в этой книге в предыдущих десяти главах, было обычным "автономным" приложением, вся программная логика которого целиком содержались в одном выполняемом файле (\*.exe). Однако одной из главных задач платформы .NET является *многократное использование программного кода*, при котором приложения могут использовать типы, содержащиеся в различных внешних компоновочных блоках (называемых также библиотеками программного кода). Целью этой главы будет подробное рассмотрение вопросов создания, инсталляции и конфигурации компоновочных блоков .NET.

Сначала мы рассмотрим различия между одномодульными и многомодульными компоновочными блоками, а также между "приватными" и "общедоступными" компоновочными блоками. Затем мы выясним, как среда, выполнений .NET определяет параметры размещения компоновочного блока, и попытаемся понять роль GAC (Global Assembly Cache – глобальный кэш компоновочных блоков), файлов конфигурации приложения (файлы \*.config), политики публикации компоновочных блоков и пространства имен System.Configuration.

**Роль компоновочных блоков .NET**

Приложения .NET строятся путем связывания произвольного числа *компоновочных блоков.*С точки зрения упрощенного подхода компоновочный блок является двоичным файлом, включающим свое описание, снабженным номером версии и поддерживаемым средой CLR (Common language Runtime – общеязыковая среда выполнения). Несмотря на тот факт, что компоновочные блоки  .NET имеют такие же расширения (\*.exe или \*.dll), как и другие двоичные файлы Win32 (включая все еще используемые серверы COM), по сути, компоновочные блоки .NET имеют с ними очень мало общего. Поэтому для начала мы рассмотрим некоторые преимущества, обеспечиваемые форматом компоновочного блока.

**Расширенные возможности многократного использования программного кода**

При построений консольных приложений в предыдущих главах могло показаться, что в создаваемом вами выполняемом компоновочном блоке содержатся *все*функциональные возможности соответствующего приложения. На самом же деле ваши приложения использовали множество типов, предлагаемых всегда доступной библиотекой программного кода .NET, mscorlib.dll (напомним, что компилятор C# ссылается на mscorlib.dll автоматически), а также библиотекой System.Windows.Forms.dll.

Вы, наверное, знаете, что *библиотека программного кода*(также называемая *библиотекой классов)*представляет собой файл \*.dll, содержащий типы, доступные для использования внешними приложениями. При создании выполняемого компоновочного блока используются компоновочные блоки, предлагаемые системой, а также пользовательские компоновочные блоки. При этом файл библиотеки программного кода не обязательно имеет вид \*.dll, поскольку выполняемый компоновочный блок может использовать и типы, определенные во внешнем выполняемом файле. В этой связи файл \*.exe тоже можно считать "библиотекой программного кода".

**Замечание.**До появления Visual Studio 2005 единственную возможность сослаться на выполняемую библиотеку программного кода обеспечивал флаг /reference компилятора C#. Но теперь ссылаться на компоновочные блоки \*.exe позволяет и диалоговое окно Add Reference (Добавление ссылки) в Visual Studio 2005.

Независимо от того, как упакована библиотека программного кода, платформа .NET позволяет использовать типы в независимой от языка форме. Например, можно создать библиотеку программного кода в C# и использовать эту библиотеку в любом другом языке программирования .NET. При этом можно не только создавать экземпляры типов в рамках других языков, но и получить производные таких типов. Базовый класс, определенный в C#, можно расширить с помощью класса, созданного в Visual Basic .NET. Интерфейсы, определенные в Pascal .NET, могут реализовываться структурами, определенными в C#. Смысл в том, что при разделении единого и монолитного выполняемого программного кода на множество компоновочных блоков .NET вы получаете *языково-нейтральную*форму программного кода, пригодного для многократного использования.

**Установка четких границ типов**

Из главы 3 вы узнали о формальных понятиях, лежащих в основе любого пространства имен .NET. Напомним, что *абсолютное имя*типа строится путем добавления префикса пространства имен (например, System) к имени типа (например, Console). Однако, строго говоря, компоновочный блок, содержащий данный тип, задает параметры дальнейшей идентификации типа. Например, если у вас есть два компоновочных блока с разными названиями (скажем, MyCars.dll и YourCars.dll), которые определяют пространство имен (CarLibrary), содержащее класс SportsCar, то эти классы во "вселенной" .NET будут считаться разными типами.

**Управление версиями**

Компоновочным блокам .NET назначается состоящий из четырех частей числовой идентификатор версии, имеющий вид *‹главный номер версии›.‹дополнительный номер версии›.‹номер компоновки›.‹номер варианта›*(если вы не укажете явно идентификатор версии с помощью свойства [AssemblyVersion], компоновочный блок автоматически получит идентификатор версии 0.0.0.0). Этот идентификатор в совокупности с необязательным значением открытого ключа позволяет множеству версий одного и того же компоновочного блока сосуществовать на одной и той нее машине в полной гармонии, Компоновочные блоки, обеспечивающие информацию об открытом ключе, называются *строго именованными.*Как будет показано в этой главе позже, при наличии строго заданного имени среда CLR способна гарантировать, что по запросу вызывающего клиента будет загружена именно та версия компоновочного блока, которая требуется.

**Самоописание**

Компоновочные блоки считаются единицами с частичным самоописанием, поскольку в них содержится информация о внешних компоновочных блоках, необходимых для правильного функционирования компоновочного блока. Так что если вашему компоновочному блоку требуются System.Windows.Forms.dll и System. Drawing.dll, то информация о них будет записана в *манифест*компоновочного блока. Вспомните из главы 1, что манифест – это блок метаданных, описывающих сам компоновочный блок (имя, версия, информация о внешних компоновочных блоках и т.д.).

Кроме данных манифеста, компоновочный блок содержит метаданные, описывающие структуру каждого содержащегося типа (имена членов, реализуемые интерфейсы, базовые классы, конструкторы и т.д.). И поскольку компоновочный блок документируется настолько "красноречиво", среда CLR не *обращается*к реестру системы Win32 для выяснения размещения компоновочного блока (что принципиально отличается от предлагавшейся ранее Microsoft модели программирования COM). Из этой главы вы узнаете, что среда CLR использует совершенно новую схему получения информации о размещении внешних библиотек программного кода.

**Средства конфигурации**

Компоновочные блоки можно инсталлировать как "приватные" или как "общедоступные". Приватные компоновочные блоки размещаются в том же каталоге (или, возможно, подкаталоге), что и использующее их приложение-клиент. Общедоступные компоновочные блоки, напротив, являются библиотеками, доступными для многих приложений, и такие компоновочные блоки устанавливаются в специальный каталог, имеющий специальное название – *глобальный кэш компоновочных блоков*(CAG).

Независимо от вида инсталляция компоновочных блоков, вы можете создавать для них XML-файлы конфигурации. С помощью этих файлов можно дать среде CLR "указание" о том, где следует искать компоновочные блоки, какую версию соответствующего компоновочного блока следует загрузить для конкретного клиента, к какому каталогу на локальной машине, в вашей локальной сети или по какому заданному адресу URL в Web следует обратиться. Более подробную информацию о XML-файлах конфигурации вы получите в дальнейшем при изучении материала этой главы.

**Формат компоновочного блока  .NET**

Теперь, когда вы знаете о некоторых преимуществах, обеспечиваемых компоновочными блоками .NET, давайте немного сместим акценты и попытаемся понять то, как устроены компоновочные блоки. С точки зрения внутренней структуры, компоновочный блок .NET (\*.dll или \*.exe) состоит из следующих элементов.

• Заголовок Win32

• Заголовок CLR

• CIL-код

• Метаданные типа

• Манифест компоновочного блока

• Необязательные встроенные ресурсы

Первые два элемента (заголовки Win32 и CLR) – это блоки данных, которыми вы можете обычно пренебречь, так что в отношении этих заголовков здесь предлагается только самая общая информация. С учетом этого мы и рассмотрим все указанные элементы по очереди.

**Заголовок Win32**

Заголовок Win32 декларирует, что компоновочный блок может загружаться и управляться средствами операционных систем семейства Windows. Данные этого заголовка также идентифицируют тип приложения (консольное, с графическим интерфейсом или библиотека программного кода \*.dll). Чтобы увидеть информацию заголовка Win32 компоновочного блока, откройте компоновочный блок .NET с помощью утилиты dumpbin.exe (в окне командной строки .NET Framework 2.0 SDK) с флагом /headers. На рис. 11.1 показана часть информации заголовка Win32 для компоновочного блока CarLibrary.dll, который вы построите в этой главе немного позже.

**Заголовок CLR**

Заголовок CLR- это блок данных, который должны поддерживать все файлы .NET (и действительно поддерживают, благодаря компилятору C#), чтобы среда CLR имела возможность обрабатывать их. По сути, этот заголовок определяет множество флагов, позволяющих среде выполнения выяснить структуру данного управляемого файла. Например, существуют флаги, позволяющие идентифицировать размещение метаданных и ресурсов в файле, выяснить версию среды выполнения, для которой создавался компоновочный блок, значение (необязательного) открытого ключа и т.д. Если с dumpbin.exe использовать флаг /clrheader, вы получите внутреннюю информацию заголовка CLR для данного компоновочного блока .NET, как показано на рис. 11.2.

Заголовок CLR компоновочного блока представляется неуправляемой структурой C-типа (IMAGE \_ COR20 \_ HEADER), определенной в файле C-заголовка corhdr.h.



Рис. 11.1. Информация заголовка Win32 компоновочного блока

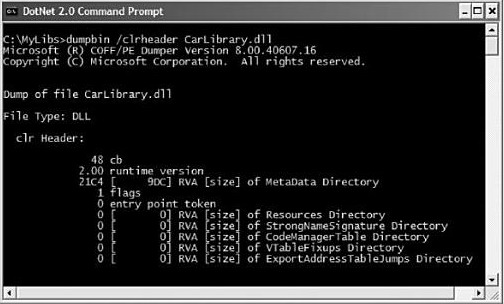


Рис. 11.2. Информация заголовка CLR компоновочного блока

Для заинтересованных читателей предлагаем ознакомиться с видом структуры, о которой здесь идет речь.

**// Структура заголовка CLR 2.0.**

 typedef struct IMAGE\_COR20\_HEADER {

**// Версии заголовка.**

 ULONG cb;

 USHORT MajorRuntimeVersion;

 USHORT MinorRuntimeVersion;

**// Таблица символов и начальная информация.**

 IMAGE\_DATA\_DIRECTORY MetaData;

 ULONG Flags;

 ULONG EntryPointToken;

**// Информация связывания.**

 IMAGE\_DATA\_DIRECTQRY Resources;

 IMAGE\_DATA\_DIRECTORY StrongNameSignature;

**// Стандартная информация адресации и связывания.**

 IMAGE\_DATA\_DIRECTQRY CodeManagerTable;

 IMAGE\_DATA\_DIRECTORY VTableFixups;

 IMAGE\_DATA\_DIRECTORY ExportAddressTableJumps;

**// Информация прекомпилированного образа (только для**

**// внутреннего использования – обнуляется)**

 IMAGE\_DATA\_DIRECTORY ManagedNativeHeader;

} IMAGE\_COR20\_HEADER;

Снова обращаем ваше внимание на то, что вам, как разработчику .NET-приложений, не придется иметь дело с информацией заголовков Win32 и CLR (за исключением того случая, когда вы захотите построить новый управляемый компилятор). Вам достаточно понимать, что каждый компоновочный блок .NET обязательно содержит эти данные, используемые средой выполнения .NET и операционной системой Win32.

**Программный код CIL, метаданные типа и манифест компоновочного блока**

В своей базе компоновочный блок содержит программный код CIL, который, как вы помните, является промежуточным языком, не зависящим от платформы и процессора. В среде выполнения внутренний CIL-код компилируется "на лету" (с помощью JIT-компилятора [just-in-time compiler – оперативный компилятор]) в специфические для данной платформы и данного процессора инструкции. В рамках такого подхода компоновочные блоки .NET действительно могут выполняться в условиях самого широкого разнообразия архитектур, устройств и операционных систем. Вы можете вполне обойтись и без понимания особенностей языка программирования CIL, но, тем не менее, в главе 15 предлагается краткое введение в синтаксис и семантику CIL.

Компоновочный блок также содержит метаданные, полностью описывающие форматы содержащихся в компоновочном блоке типов, а также форматы внешних типов, на которые ссылается данный компоновочный блок. Среда выполнения .NET использует эти метаданные для нахождения типов (и их членов) в бинарном файле, для размещения типов в памяти удаленного вызова методов. Детали формата метаданных .NET будут изучаться в главе 12 при рассмотрении сервисов отображения.

Кроме того, компоновочный блок должен содержать ассоциированный *манифест*(также называемый *метаданными компоновочного блока*). Манифест документирует все модули данного компоновочного блока, задает версию компоновочного блока, а также предлагает информацию обо всех внешних компоновочных блоках, да которые ссылается данный компоновочный блок (в отличие от библиотек COM, которые не предлагают способа документирования внешних зависимостей). В процессе изучения материала этой главы вы поймете, что среда CLR интенсивно использует манифест компоновочного блока при определении внешних ссылок.

**Замечание**. К этому моменту, наверное, уже не нужно повторять, что для просмотра программного кода CIL компоновочного блока, метаданных типов иди манифеста можно использовать ildasm.exe. Я предполагаю, что вы будете часто использовать ildasm.exe при изучении примеров этой главы.

**Необязательные ресурсы компоновочного блока**

Наконец, компоновочный блок .NET может содержать любой набор встроенных ресурсов, таких как, например, пиктограммы приложении, графические файлы, звуковые фрагменты или таблицы строк. Платформа .NET обеспечивает поддержку *сопутствующих компоновочных блоков,*которые не содержат ничего, кроме локализованных ресурсов. Это может понадобиться тогда, когда требуется предоставить ресурсы на разных языках (английском, немецком и т.д.) при создании программного обеспечения, используемого в разных странах. Тема создания сопутствующих компоновочных блоков выходит за рамки этой книги, но при изучении GDI+ в главе 20 вы узнаете, как встроить ресурсы приложения в компоновочный блок.

**Одномодульные и многомодульные компоновочные блоки**

Компоновочный блок можно скомпоновать из одного или нескольких *модулей.*Модуль – это просто обобщающий термин для обозначения двоичных файлов .NET. В большинстве случаев компоновочный блок компонуется из одного модуля. В таком случае наблюдается взаимно однозначное соответствие между (логическим) компоновочным блоком и лежащим в его основе (физическим) двоичным файлом (отсюда и появляется термин *одномодульный компоновочный блок*).

Одномодульные компоновочные блоки содержат все необходимые элементы (информация: заголовка, программный код CIL, метаданные типов, манифест и требуемые ресурсы) в одном пакете \*.exe или \*.dll. На рис. 11.3 показана композиционная схема одномодульного компоновочного блока.

Многомодульный компоновочный блок, напротив, является набором .NET-файлов \*.dll, которые инсталлируются как одна логическая единица и контролируются по единому идентификатору версии. Формально один из этих файлов \*.dll называется *первичным модулем,*он содержит *манифест компоновочного блока*(а также необходимый программный код CIL, метаданные, информацию заголовка и опциональные ресурсы). Манифест первичного модуля содержит записи о каждом из связанных файлов \*.dll, от которых он зависит.



Рис. 11.3. Одномодульный компоновочный блок

По соглашению о выборе имен вторичные модули в многомодульном компоновочном блоке имеют расширение \*.netmodule, однако это не является непременным требованием CLR. Вторичные файлы \*.netmodule также содержат CIL-код и метаданные типов, а также *манифест уровня модуля,*в котором просто записана информация о внешних компоновочных блоках, необходимых для данного конкретного модуля.

Главное преимущество построения многомодульных компоновочных блоков заключается в том, что они обеспечивают очень эффективный способ загрузки содержимого. Предположим, например, что у нас есть машина, которая ссылается на удаленный многомодульный компоновочный блок, состоящий из трех модулей, причем первичный модуль установлен на машине клиента. Если клиент потребует тип из вторичного удаленного файла \*.netmodule, среда CLR загрузит двоичный выполняемый файл на локальную машину по требованию в специальное место, называемое *кэшем загрузки.*Если каждый файл \*.netmodule имеет объем 1Мбайт, я уверен, вы поймете, в чем здесь преимущество.

Другим преимуществом многомодульных компоновочных блоков является то, что для них можно создавать модули на разных языках программирования .NET (что очень удобно в больших корпорациях, где разные подразделения могут отдавать предпочтение разным языкам .NET). После компиляции отдельных модулей эти модули можно логически "связать" в один компоновочный блок, используя, например, такой инструмент, как компоновщик (al.exe).

В любом случае следует понимать, что модули, которые образуют многомодульный компоновочный блок, *не*связаны непосредственно в один (больший) файл. Скорее, многомодульные компоновочные блоки связаны только логически информацией, содержащейся в манифесте первичного модуля. На рис. 11.4 показана схема многомодульного компоновочного блока, состоящего из трех модулей, каждый из которых написан на своем языке программирования .NET.

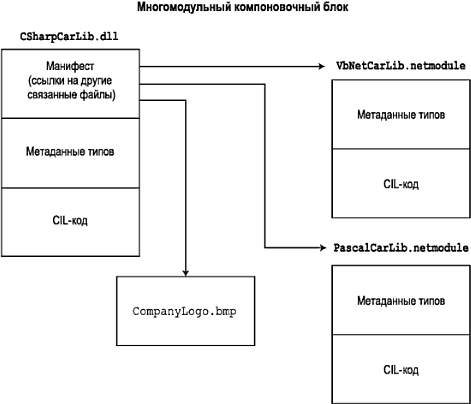


Рис. 11.4. Первичный модуль записывает информацию о вторичных модулях в манифест компоновочного блока

К этому моменту вы (я надеюсь) уже лучше понимаете внутреннюю структур двоичного файла .NET. Теперь "погрузимся" в обсуждение подробностей процесса построения и выбора конфигурации библиотек программного кода.

**Создание и использование одномодульных компоновочных блоков**

Чтобы инициировать процесс понимания компоновочных блоков .NET, мы с вами создадим одномодульный компоновочный блок \*.dll (с именем CarLibrary), содержащий небольшой набор открытых типов. Чтобы построить библиотеку программного кода в Visual Studio 2005, выберите рабочую область Class Library (Библиотека классов) в окне Создания проектов (рис. 11.5).

Процесс разработки нашей библиотеки мы начнем с создания абстрактного базового класса Car (автомобиль), определяющего ряд защищенных членов-данных, доступных через пользовательские свойства.

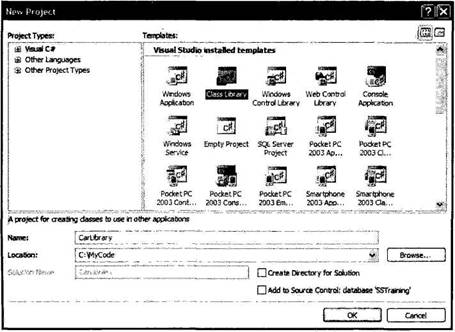


Рис. 11.5. Создание библиотеки программного кода C#

Этот класс имеет один абстрактный метод TurboBoost(), в котором используется пользовательский перечень (EngineState), представляющий текущее состояние двигателя автомобиля.

using System;

namespace CarLibrary {

**// Представляет состояние двигателя.**

 public enum EngineState { engineAlive, engineDead }

**// Абстрактный базовый класс в данной иерархии.**

 public abstract class Car {

  protected string petName;

  protected short currSpeed;

  protected short maxSpeed;

  protected EngineState egnState = EngineState.engineAlive;

  public abstract void TurboBoost();

  public Car(){}

  public Car(string name, short max, short curr) {

   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; } }

 }

}

Теперь предположим, что у вас есть два прямых "наследника" типа Car, имена которых MiniVan (минивэн) и SportsCar (спортивный автомобиль). Каждый из них подходящим образом переопределяет абстрактный метод TurboBoost().

using System;

using System.Windows.Forms;

namespace CarLibrary {

 public class SportsCar: Car {

  public SportsCar(){}

  public SportsCar(string name, short max*,*short curr): base(name, max, curr) {}

  public override void TurboBoost() {

   MessageBox.Show("Скорость черепахи!", "Побыстрее бы…");

  }

 }

 public class MiniVan: Car {

  public MiniVan(){}

  public MiniVan(string name, short max, short curr): base(name, max, curr){}

  public override void TurboBoost() {

**// Минивэн с турбонаддувом встретишь не часто!**

   egnState = EngineState.engineDead;

   MessageBox.Show("Звoните в автосервис!", "Машина сломалась…");

  }

 }

}

Обратите внимание на то, что каждый из подклассов реализует TurboBoost() с помощью класса MessageBox, определенного в компоновочном блоке System. Windows.Forms.dll. Чтобы наш компоновочный блок мог использовать типы, определенные в рамках этого внешнего компоновочного блока, для проекта CarLibrary нужно указать ссылку на соответствующий двоичный файл в диалоговом окне Add Reference (Добавление ссылки), доступном в Visual Studio 2005 при выборе Project→Add Reference из меню (рис. 11.6).

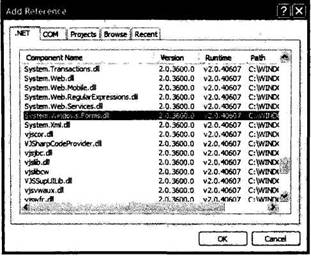


Рис. 11.6. Здесь добавляются ссылки на внешние компоновочные блоки .NET

Очень *важно*понимать, что в списке компоновочных блоков диалогового окна Add Reference могут быть представлены не все компоновочные блоки .NET, имеющиеся на вашей машине. Диалоговое окно Add Reference *не* отображает созданные вами пользовательские компоновочные блоки и *не*отображает компоновочные блоки, размещенные в GAC. Это диалоговое окно предлагает список общих компоновочных блоков, на отображение которых запрограммирована система Visual Studio 2005. При построении приложения, для которого требуется компоновочный блок, не представленный в списке диалогового окна Add Reference, вам придется перейти на вкладку Browse (Просмотр) и вручную найти необходимый файл \*.dll или \*.exe.

**Замечание.**Можно сделать так, чтобы пользовательские компоновочные блоки тоже появлялись в списке диалогового окна Add Reference, если установить их копии в папку C:\Program Files\Microsoft Visual Studio 8\Common7\lDE\PublicAssemblies, но большого смысла в этом нет. На вкладке Recent (Недавние ссылки) предлагается список компоновочных блоков, на которые вы недавно ссылались.

**Анализ манифеста**

Перед тем как использовать CarLibrary.dll в приложении-клиенте, давайте выясним, из чего скомпонована библиотека программного кода. Предположив, что наш проект уже скомпилирован, загрузим CarLibrary.dll в ildasm.exe (рис. 11.7).

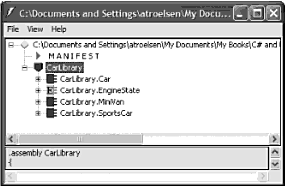


Рис. 11.7. Библиотека CarLibrary.dll в окне ildasm.exe

Теперь откройте манифест файла CarLibrary.dll двойным щелчком на пиктограмме MANIFEST. В первом блоке программного кода манифеста указываются внешние компоновочные блоки, необходимые соответствующему компоновочному блоку для правильного функционирования. Как вы помните, CarLibrary.dll использует типы из mscorlib.dll и System.Windows.Forms.dll, и оба эти файла будут указаны в списке манифеста с помощью лексемы .assembly extern внешних связей компоновочного блока.

**.assembly extern**mscorlib {

 .publickeytoken = (В7 7A 5C 56 19 34 E0 89)

 .ver 2:0:0:0

}

**.assembly extern**System.Windows.Forms {

 .publickeytoken = (B7 7A 5C 56 19 34 E0 89)

 .ver 2:0:0:0.

}

Здесь каждый блок .assembly extern снабжен директивами .publickeytoken и .ver. Инструкция .publickeytoken указывается только тогда, когда компоновочный блок имеет *строгую форму имени*(подробности будут приведены в этой главе позже). Лексема.ver обозначает (конечно же) числовой идентификатор версии.

После списка каталогизации внешних ссылок вы обнаружите ряд лексем.custom, идентифицирующих атрибуты уровня компоновочного блока. Проверив файл AssemblyInfо.cs, созданный в Visual Studio 2005, вы обнаружите, что эти атрибуты представляют такую информацию о компоновочном блоке, как название компании, торговая марка и т.д. (все соответствующие поля в данный момент пусты). В главе 14 атрибуты будут рассматриваться подробно, поэтому пока что не обращайте на них большого внимания. Однако следует знать, что атрибуты из AssemblyInfo.cs добавляют в манифест ряд лексем .custom, например, [AssemblyTitle].

.assembly CarLibrary {

 …

 .custom instance void [mscorlib]

 System.Reflection.**AssemblyTitleAttribute**::.ctor(string) = (01 00 00 00 00)

 .hash algorithm 0x00008004

**.ver 1:0:454:30104**

}

**.module CarLibrary.dll**

Наконец, вы можете заметить, что лексема .assembly используется для обозначения понятного имени компоновочного блока (CarLibrary), в то время как лексема .module указывает имя самого модуля (CarLibrary.dll). Лексема .ver определяет номер версии, назначенный для компоновочного блока в соответствии с атрибутом [AssemblyVersion] из AssemblyInfo.cs. Подробнее об управлении версиями компоновочного блока будет говориться в этой главе позже, а сейчас необходимо заметить, что групповой символ \* в атрибуте [AssemblyVersion] информирует Visual Studio 2005 о необходимости в процессе компиляции выполнить приращение для идентификатора версии в отношении номеров компоновки и варианта.

**Анализ CIL-кода**

Напомним, что компоновочный блок не содержит специфических для платформы инструкций, а содержит независимый от платформы CIL-код. Когда среда выполнения .NET загружает компоновочный блок в память, этот CIL-код компилируется (с помощью JIT-компилятора) в инструкции, понятные для данной платформы. Если выполнить двойной щелчок на строке метода TurboBoost() класса SportsCar, с помощью ildasm.exe откроется новое окно, в котором будут показаны CIL-инструкции.

.method public hidebysig virtual instance void TurboBoost() cil managed {

 // Code size 17 (0x11)

 .maxstack 2

 IL\_0000: ldstr "Ramming speed!"

 IL\_0005: ldstr "Faster is better…"

 IL\_000a: call valuetype [System.Windows.Forms] System.Windows.Forms.DialogResult [System.Windows.Forms] System.Windows.Forms.MessageBox::Show(string, string)

 IL\_000f: pop

 IL\_0010: ret

} // end of method SportsCar::TurboBoost

Обратите внимание на то, что для идентификации метода, определенного типом SportsCar, используется лексема .method. Члены-переменные, определенные типом, обозначаются лексемой .field. Напомним, что класс Car определяет набор защищенных данных, например, таких как currSpeed.

.field family int 16 currSpeed

Свойства обозначены лексемой.property. Этот CIL-код описывает открытое свойство CurrSpeed (заметьте, что характеристики read/write свойства обозначаются лексемами .get и .set).

**.property**instance int16 CurrSpeed() {

**.get**instance int16 CarLibrary.Car::get\_CurrSpeed()

**.set**instance void CarLibrary.Car::set\_CurrSpeed(int16)

} // end of property Car::CurrSpeed

**Анализ метаданных типов**

Наконец, если вы сейчас нажмете комбинацию клавиш ‹Ctrl+M›, ildasm.exe отобразит метаданные для каждого из типов, имеющихся в компоновочном блоке CarLibrary.dll (рис. 11.8).

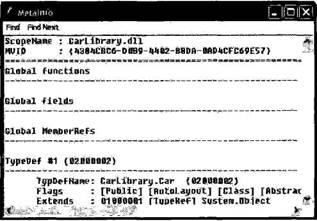


Рис. 11.8. Метаданные для типов на CarLibrary.dll

Теперь, после того как мы с вами заглянули внутрь компоновочного блока CarLibrary.dll, мы можем приступить в построению приложений клиента.

**Исходный код**. Проект CarLibrary размещен в подкаталоге, соответствующем главе 11.

**Создание приложения-клиента в C#**

Ввиду тот, что все типы CarLibrary были объявлены с ключевым словом public, другие компоновочные блоки способны их использовать. Напомним, что вы можете также объявлять типы с использованием ключевого слова C# internal (именно этот режим доступа в C# используется до умолчанию, когда вы определяете тип без указания public). Внутренние типы могут использоваться только тем компоновочным блоком, в котором они определены. Внешние клиенты не могут ни видеть, ни создавать внутренние типы компоновочных блоков.

**Замечание**. Сегодня .NET 2.0 предлагает возможность указать "дружелюбные" компоновочное блоки, которые позволяют использовать свои внутренние типы заданным компоновочным блокам. Подробности можно найти в разделе документации .NET Framework 2.0 SDK с описанием класса InternalsVisibleToAttribute.

Для использования открытых типов CarLibrary создайте новый проект консольного приложения C# (CSharpCarClient). После этого добавьте ссылку на Carbibrary.dll на вкладке Browse диалогового окна Add Reference (если вы скомпилировали CarLibrary.dll в Visual Studio 2005, ваш компоновочный блок будет размещен в подкаталоге \Bin\Debug папки проекта CarLibrary). После щелчка на кнопке ОК Visual Studio 2005 поместит копию CarLibrary.dll в папку \Bin\Debug папки проекта CSharpCarClient (рис. 11.9).



Рис. 11.9. Visual Studio 2005 копирует приватные компоновочные блоки в каталог клиента

С этого момента вы можете компоновать приложение-клиент с использованием внешних типов. Модифицируйте свой исходный C#-файл так.

using System;

**// Не забудьте 'использовать' пространство имен CarLibrary!**

using CarLibrary;

namespace CSharpCarClient {

 public class CarClient {

  static void Main(string[] args) {

**// Создание спортивной машины.**

   SportsCar viper = new SportsCar("Viper", 240, 40);

   viper.TurboBoost();

**// Создание минивэна.**

   MiniVan mv = new MiniVan();

   mv.TurboBoost();

   Console.ReadLine();

  }

 }

}

Этот программный код очень похож на программный код приложений, создававшихся нами ранее. Единственным отличием является то, что теперь приложение-клиент C# использует типы, определенные в отдельном пользовательском компоновочном блоке. Запустите эту программу, и вы увидите на своем экране целый ряд окон сообщений.

**Исходный код**. Проект CSharpCarClient размещён в подкаталоге, соответствующем главе 11.

**Создание приложения-клиента в Visual Basic .NET**

Чтобы продемонстрировать языковую независимость платформы .NET, создадим другое консольное приложение (VbNetCarClient) на этот раз с помощью Visual Basic .NET (рис. 11.10). Создав проект, укажите ссылку на CarLibrary.dll с помощью диалогового окна Add Reference.

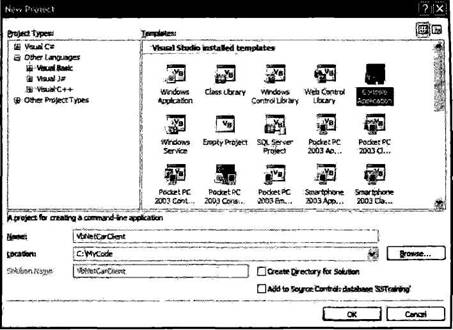


Рис. 11.10. Создание консольного приложения Visual Basic .NET

Как и в C#, в Visual Basic .NET требуется указать список всех пространств имен, используемых в текущем файле. Но в Visual Basic .NET для этого предлагается использовать ключевое слово Imports, а не ключевое слово using, как в C#. С учетом этого добавьте следующий оператор Imports в файл программного кода Module1.vb.

Imports CarLibrary

Module Module1

 Sub Маin()

 End Sub

End Module

Обратите внимание на то, что метод Main() определен в рамках типа Module Visual Basic .NET (который не имеет ничего общего с файлами \*.netmodule многомодульных компоновочных блоков). В Visual Basic .NET Module используется просто для обозначения определения изолированного класса, содержащего только статические методы. Чтобы сделать это утверждение более понятным, вот аналог соответствующей конструкции в C#.

**// 'Module' в VB .NET - это просто изолированный класс,**

**// содержащий статические методы.**

public sealed class Module1 {

 public static void Main() {

 }

}

Так или иначе, чтобы использовать типы MiniVan и SportsCar в рамках синтаксиса Visual Basic .NET, измените метод Main() так, как предлагается ниже.

Sub Main()

 Console.WriteLine("\*\*\*\*\* Забавы с Visual Basic .NET \*\*\*\*\*")

 Dim myMiniVan As New MiniVan()

 myMiniVan.TurboBoost()

 Dim mySportsCar As New SportsCar()

 mySportsCar.TurboBoost()

 Console.ReadLine()

End Sub

После компиляции и выполнения приложения вы снова увидите соответствующий набор окон с сообщениями.

**Межъязыковое перекрестное наследование**

Весьма привлекательной возможностью .NET является *межъязыковое перекрестное наследование*. Для примера давайте создадим новый класс Visual Basic .NET, который будет производным от SportsCar (напомним, что последний был создан в C#). Сначала добавим файл нового класса с именем PerformanceCar.vb в имеющееся приложение Visual Basic .NET (с помощью выбора Project→Add Class из меню). Обновим исходное определение класса путем получения производного типа из SportsCar, используя ключевое слово Inherits. Кроме того, переопределим абстрактный метод TurboBoost(), используя для этого ключевое слово Overrides.

Imports CarLibrary

' **Этот VB-тип является производным C#-типа SportsCar.**

Public Class PerformanceCar Inherits SportsCar

 Public Overrides Sub TurboBoost()

  Console.WriteLine("От нуля до 100 за какие-то 4,8 секунды…")

 End Sub

End Class

Чтобы проверить работу нового типа класса, обновите метод Main() модуля так.

Sub Main()

 …

 Dim dreamCar As New PerformanceCar()

**' Наследуемое свойство.**

 dreamCar.PetName = "Hank"

 dreamCar.TurboBoost()

 Console.ReadLine()

End Sub

Обратите внимание на то, что объект dreamCar способен вызывать любые открытые члены (например, свойство PetName) по цепочке наследования, несмотря на то, что базовый класс определен на совершенно другом языке и в другой библиотеке программного кода.

**Исходный код**. Проект VbNetCarClient размещен в подкаталоге, соответствующем Главе 11.

**Создание и использование многомодульных компоновочных блоков**

Теперь, когда вы научились строить и использовать одномодульные компоновочные блоки, рассмотрим процесс создания многомодульных компоновочных блоков, Напомним, что многомодульный компоновочный блок – это просто набор связанных модулей, инсталлируемых как цельная единица и контролируемых по единому номеру версии. На момент создания этой книги в Visual Studio 2005 не предлагался шаблон проекта дли многомодульного компоновочного блока C#. Поэтому для построения такого проекта вам придется использовать компилятор командной строки (csc.exe).

Для примера мы с вами построим многомодульный компоновочный блок с названием AirVehicles (авиатранспорт). Первичный модуль (airvehicles.dll) будет содержать один тип класса Helicopter (вертолет). Соответствующий манифест (также содержащийся в airvehicles.dll) каталогизирует дополнительный файл \*.netmodule с именем ufo.netmodule, который будет содержать другой тип класса, называющийся, конечно же, Ufo (НЛО). Эти два типа класса физически содержатся в отдельных двоичных файлах, но мы сгруппируем их в одном пространстве имен, названном AirVehicles. Наконец, оба класса будут созданы с помощью C# (хотя вы, если хотите, можете использовать и разные языки).

Для начала откройте простой текстовый редактор (например, Блокнот) и создайте следующее определение класса Ufo, сохранив затем его в файле с именем ufo.cs.

using System;

namespace AirVehicles {

 public class Ufo {

  public void AbductHuman() {

   Console.WriteLine("Сопротивление бесполезно");

  }

 }

}

Чтобы скомпилировать этот класс в .NET-модуль, перейдите в папку, содержащую ufo.cs. и введите следующую команду компилятору C# (опция module флага /target "информирует" csc.exe о том, что необходимо построить файл \*.netmodule, а не \*.dll или \*.exe).

csc.exe /t:module ufo.cs

Если теперь заглянуть в папку, содержащую файл ufo.cs, вы должны увидеть новый файл с именем ufo.netmodule (проверьте!). После этого создайте новый файл с именем helicopter.cs, содержащий следующее определение класса.

using System;

namespace AirVehicles {

 public class Helicopter {

  public void TakeOff() {

   Console.WriteLine("Вертолет на взлет!");

  }

 }

}

Поскольку название airvehicles.dll было зарезервировано для первичного модуля нашего многомодульного компоновочного блока, вам придется компилировать helicopter.cs с использованием опций /t:library и /out:. Чтобы поместить запись о двоичном объекте ufo.netmodule в манифест компоновочного блока, вы должны также указать флаг /addmodule. Все это делает следующая команда.

csc /t:library /addmodule:ufo.netmodule /out:airvehicles.dll helicopter.cs

К этому моменту ваш каталог должен содержать первичный модуль airvehicles.dll, а также вторичный ufo.netmodule.

**Анализ файла ufo.netmodule**

Теперь с помощью ildasm.exe откройте ufo.netmodule. Вы убедитесь, что \*.netmodule содержит *манифест уровня модуля,*однако его единственной целью является указание списка всех внешних компоновочных блоков, на которые есть ссылки в соответствующем программном коде. Поскольку класс Ufo, по сути, выполняет только вызов Console.WriteLine(), вы обнаружите следующее.

.assembly extern mscorlib {

 .publickeytoken = (B7 7A 5C 56 19 34 E0 89)

 .ver 2:0:0:0

}

.module ufo.netmodule

**Анализ файла airvehicles.dll**

Теперь в помощью ildasm.exe откройте первичный модуль airvehicles.dll и рассмотрите манифест уровня компоновочного блока. Вы увидите, что лексемы.file документируют ассоциированные модули многомодульного компоновочного блока (в данном случае ufo.netmodule). Лексемы.class extern используются для указания имен внешних типов из вторичного модуля (Ufo)*,*на которые имеются ссылки.

.assembly extern mscorlib {

 .publickeytoken = (B7 7A 5C 56 19 34 E0 89)

 .ver 2:0:0:0

}

.assembly airvehiсles {

 …

 .hash algorithm 0x00008004

 .ver 0:0:0:0

}

**.file ufо**.**netmodule**

…

.class extern public AirVehicles.Ufo {

 .file ufo.netmodule

 .class 0x02000002

}

.module airvehicles.dll

Снова подчеркнем, что манифест компоновочного блока является единственным объектом, связывающим airvehicles.dll и ufo.netmodule. Указанные два бинарных файла не содержатся в одном, большем \*.dll.

**Использование многомодульного компоновочного блока**

Пользователей многомодульного компоновочного блока не должно заботить то, что компоновочный блок, на который они ссылаются, состоит из нескольких модулей. Чтобы пояснить ситуацию, мы построим новое приложение-клиент Visual Basic .NET с командной строки. Создайте новый файл Client.vb, содержащий приведенное ниже определение. Сохраните его в там месте, где находится ваш многомодульный компоновочный блок.

Imports AirVehicles

 Module Module1

 Sub Main()

  Dim h As New AirVehicles.Helicopter()

  h.Takeoff()

  ' **Это загрузит \*.netmodule по требованию.**

  Dim u As New UFO()

  u.AbductHuman()

  Console.ReadLine()

 End Sub

End Module

Чтобы скомпилировать этот выполняемый компоновочный блок с командной строки, используйте компилятор командной строки Visual Basic .NET vbc.exe со следующим набором команд.

vbc /r:airvehicles.dll *\*.*vb

Обратите внимание на то, что при ссылке на многомодульный компоновочный блок компилятору нужно указать только имя первичного модуля (файлы \*.netmodule загружаются по запросу программного кода клиента). В самих файлах \*.netmodules нет индивидуального номера версии, и они не могут непосредственно загружаться средой CLR Файл \*. netmodule может загружаться только первичным модулем (например, файлом, содержащим манифест компоновочного блока).

**Замечание.**В Visual Studio 2005 позволяется ссылаться и на многомодульные компоновочные блоки. Используйте диалоговое окно Add Reference и выберите первичный модуль, В результате будут скопированы и все связанные файлы \*.netmodule.

К этому моменту вы должны чувствовать себя вполне уверенно при построении как одномодульных, так и многомодульных компоновочных блоков. Конечно, можно утверждать, что с вероятностью 99.99 % ваши компоновочные блоки будут одномодульными. Однако и многомодульные компоновочные блоки могут оказаться полезными, если вы захотите разделить большие по объему двоичные файлы на менее объемные модульные единицы (это может пригодиться для сценариев удаленной загрузки). Следующим нашим шагом будет формализация понятия приватного компоновочного блока.

**Исходный код.**Проект MultifileAssembly размещен в подкаталоге, соответствующем главе 11.

**Приватные компоновочные блоки**

Компоновочные блоки, которые создавались вами в этой главе до сих пор, инсталлировались, как *приватные компоновочные блоки.*Приватные компоновочные блоки должны размещаться в том же каталоге, что и приложение-клиент (такой каталог называется *каталогом приложения*) или в его подкаталоге. Напомним, что результатом добавления ссылки на CarLibrary.dll при построении приложений CSharpCarClient.exe и VbNetCarClient.exe в Visual Studio 2005 было копирование CarLibrary.dll в каталог приложения-клиента.

Когда программа-клиент использует типы, определенные в этом внешнем компоновочном блоке, среда CLR просто загружает локальную копию CarLibrary.dll. Ввиду того, что среда выполнения .NET не использует реестр системы при поиске компоновочных блоков, вы можете переместить компоновочные блоки CSharpCarClient.exe (или VbNetCarClient.exe) вместе c CarLibrary.dll в другое место на своей машине и успешно запустить приложение.

Деинсталляция (a также тиражирование) приложения, использующего исключительно приватные компоновочные блоки, не требует особых усилий: нужно просто удалить (или скопировать) папку приложения. В отличие от COM-приложений, здесь не нужно беспокоиться о десятках "осиротевших" параметров, разделов и ветвей реестра. Но еще более важно то, что удаление приватных компоновочных блоков никак не влияет на работоспособность других приложений, установленных на машине.

**Идентификация приватных компоновочных блоков**

Полный идентификатор приватного компоновочного блока состоит из понятного имени компоновочного блока и числового номера его версии, которые должны быть записаны в манифест компоновочного блока. *Понятное имя*(friendly name) – это просто имя модуля, содержащего манифест компоновочного блока, без файлового раcширения. Так, если вы проверите манифест компоновочного блока CarLibrary.dll, то обнаружите там следующее (ваша версия будет, скорее всего, другой).

.assembly.CarLibrary {

 …

 .**ver** 1:0:454:30104

}

Ввиду изолированной природы приватного компоновочного блока, имеет смысл то, что среда CLR не использует номер версии при выяснении места размещения такого компоновочного блока. Предполагается, что для приватных компоновочных блоков не обязательно выполнять проверку версии, поскольку приложение-клиент является единственным приложением, "знающим" об их существовании. Поэтому вполне вероятно, что на одной машине будет много копий одного и того же приватного компоновочного блока в разных каталогах приложений.

**Процесс зондирования**

Среда выполнения .NET выясняет место размещения приватного компоновочного блока с помощью технологий *зондирования,*которые на самом деле оказываются намного менее агрессивными, чем кажется из названия. Зондирование представляет собой процесс отображения запроса внешнего компоновочного блока в соответствующее место размещения запрошенного двоичного файла, Запрос на загрузку компоновочного блока может быть либо *неявным,*либо *явным.*Неявный запрос выполняется тогда, когда среда CLR использует манифест для выяснения места расположения компоновочного блока, определенного с помощью лексемы .assembly extern.

**// Неявный запрос загрузки.**

**.assembly extern CarLibrary**

{…}

Явный запрос загрузки происходит при использовании в программе метода Load() или LoadFrom() типа System.Reflection.Assembly, обычно с целью динамического связывания и динамического вызова членов запрашиваемого типа. Мы рассмотрим эти темы позже в главе 12, а сейчас только приведем пример явного запроса загрузки в следующем фрагменте программного кода.

**// Явный запрос загрузки.**

Assembly asm = Assembly.Load("CarLibrary");

В любом из этих случаев среда CLR извлекает понятное имя компоновочного блока и начинает зондирование каталога приложения-клиента в поисках файла с именем CarLibrary.dll. Если этот файл не обнаружен, делается попытка найти выполняемый компоновочный блок с тем же понятным именем (CarLibrary.exe). Если ни одного из указанных файлов в каталоге приложения не обнаруживается, среда выполнения прекращает попытки и генерирует исключение FileNotFound.

**Замечание.**Если запрошенного компоновочного блока в каталоге приложения-клиента нет, среда CLR пытается проверить подкаталог клиента с именем, соответствующим понятному имени запрошенного компоновочного блока (скажем, C:\MyClient\CarLibrary). Если запрошенный компоновочный блок обнаружится в таком подкаталоге, среда CLR загрузит найденный компоновочный блок в память.

**Конфигурация приватных компоновочных блоков**

Конечно, можно инсталлировать .NET-приложение с помощью простого копирования всех требуемых компоновочных блоков в одну папку на жестком диске пользователя, но вы, скорее всего, предпочтете определить ряд подкаталогов для группировки взаимосвязанного содержимого. Предположим, например, что у вас есть каталог приложения C:\MyApp, содержащий CSharpCarClient.exe. В этом каталоге может быть подкаталог с именем MyLibraries, который содержит CarLibrary.dll.

Несмотря на предполагаемую связь между этими двумя каталогами, среда CLR не будет зондировать подкаталог MyLibraries, если вы не создадите файл конфигурации с соответствующим требованием. Файлы конфигурации состоят из XML-элементов, позволяющих влиять на процесс зондирования. "По закону" файлы конфигурации должны иметь то же имя, что и соответствующе приложение, но иметь расширение \*.config, и должны размещаться в каталоге приложения-клиента. Так, если вы хотите создать файл конфигурации для CSharpCarClient.exe, он должен называться CSharpCarClient.exe.config.

Для иллюстрации создайте новый каталог на вашем диске C, назвав его MyApp (например, с помощью Windows Explorer). Затем скопируйте CSharpCarClient.exe и CarLibrary.dll в этот новый каталог и запустите программу на выполнение с помощью двойного щелчка на ее файле. Ваша программа должна выполниться успешно (вспомните о том, что компоновочные блоки не требуют регистрации!). Теперь создайте в C:\MyApp подкаталог, выбрав для него название MyLibraries (рис. 11.11), и переместите в него CarLibrary.dll.

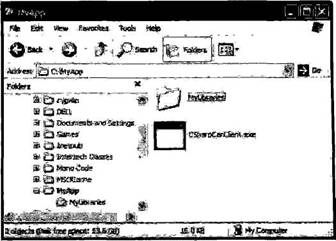


Рис. 11.11. Теперь CarLibrary.dll размещается в подкаталоге MyLibraries

Попытайтесь выполнить программу снова. Ввиду того, что среда CLR не сможет найти "CarLibrary" непосредственно в каталоге приложения, вы получите необработанное исключение FileNotFound (файл не найден).

Чтобы выправить ситуацию создайте новый файл конфигурации CSharpCarClient.exе.config и сохраните его в папке, содержащей приложение CSharpCarClient.exe (в данном случае это папка C:\MyApp). Откройте cозданный файл и введите в него следующий код в точности так, как показано ниже (язык XML является чувствительным к регистру символов).

<configuration>

 ‹runtime›

  ‹assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1"›

   ‹probing privatePath="MyLibraries"/›

  ‹/assemblyBinding›

 ‹/runtime›

‹/configuration›

Файлы \*.config .NET всегда начинаются корневым элементом ‹configuration›. Вложенный в него элемент ‹runtime› может содержать элемент ‹assemblyBinding›, который, в свою очередь, может содержать вложенный элемент ‹probing›. Для данного примера наиболее важным является атрибут privatePath, поскольку он используется для указания подкаталогов в каталоге приложения, где среда CLR должна осуществлять зондирование.

Обратите особое внимание на то, что элемент ‹probing› не указывает, какой компоновочный блок размещается в соответствующем подкаталоге. Поэтому вы не можете сказать, что "CarLibrary размещается в подкаталоге MyLibraries, a MathUtils – в подкаталоге Bin". Элемент ‹probing› просто дает среде CLR "инструкцию" при поиске запрошенного компоновочного блока исследовать указанные подкаталоги, пока не обнаружится первое совпадение.

**Замечание.**Атрибут privatePath *нельзя*использовать для указания ни абсолютного (C:\Папка\Подпапка), ни относительного (…\\ОднаПапка\\ДругаяПапка) пути! Если вы хотите указать каталог вне пределов каталога приложения-клиента, вам придется использовать другой XML-элемент – элемент ‹codeBase› (дополнительные подробности об этом элементе будут приведены в этой же главе немного позже).

Чтобы указать с помощью атрибута privatePath множество подкаталогов, используйте список значений, разделенных точками с запятой. В данный момент у вас в этом нет никакой необходимости, но вот вам пример, в котором CLR дается указание проверить подкаталоги клиента MyLibraries и MyLibraries\Tests.

‹probing privatePath="MyLibraries; MyLibraries\Tests"/›

После создания CSharpCarClient.exe.config выполните приложение-клиент с помощью двойного щелчка на выполняемом файле в программе Проводник Windows. Вы должны обнаружить, что теперь CSharpCarClient.exe выполняется без проблем (если это не так, проверьте введенные данные на отсутствие опечаток).

Затем, с целью проверки, измените (произвольным образом) имя файла конфигурации и попытайтесь выполнить программу еще раз. Приложение-клиент должно выдать отказ. Вспомните о том, что файл \*.config должен иметь префикс, соответствующий имени приложения-клиента. В качестве последней проверки откройте свой файл конфигурации для редактирования и перепишите любой из XML-элементов символами верхнего регистра. После сохранения файла выполнение вашего клиента должно стать невозможным (поскольку язык XML является чувствительным к регистру символов).

**Файлы конфигурации и Visual Studio 2005**

Вы, конечно, можете всегда создавать XML-файлы конфигурации вручную с помощью своего любимого текстового редактора, но Visual Studio 2005 позволяет создать файл конфигурации в процессе построения программы-клиента. Для примера загрузите в Visual Studio 2005 проект CSharpCarClient и добавьте в него новый элемент Application Configuration File (Файл конфигурации приложения), выбрав Project→Add New Item из меню. Перед тем как щелкнуть на кнопке ОК, обратите внимание на то, что файл получит название App.config (не переименовывайте его!). Если после этого заглянуть в окно Solution Explorer (Обозреватель решений), то вы увидите, что в текущий проект добавлен файл App.config (см. рис. 11.12).

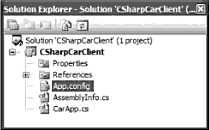


Рис. 11.12. Файл App.config в Visual Studio 2005

После этого вы сможете ввести все необходимые XML-элементы для создаваемого клиента. И здесь следует отметать кое-что действительно интересное. Каждый раз при компиляции проекта Visual Studio 2005 будет автоматически копировать данные App.config в каталог \Bin\Debug, назначая копии имя с учетом соответствующего соглашения о назначении имен (например, имя CSharpCarClient.exe.config). Однако это происходит только в том случае, когда файл конфигураций называется Арр.config. При этом вам придется поддерживать только App.config, a Visual Studio 2005 гарантирует, что каталог приложения будет содержать самые последние и самые полные данные (даже если вы, например, переименуете свой проект).

**Утилита конфигурации NET Framework 2.0**

Создание файлов \*.config вручную не является слишком большой проблемой, но, тем не менее, .NET Framework 2.0 SDK предлагает инструмент, который позволяет строить XML-файлы конфигурации в рамках графического интерфейса пользователя. Утилиту Microsoft .NET Framework 2.0 Configuration можно найти в папке Администрирование, размещенной в панели управления Windows. Запустив этот инструмент, вы увидите ряд опций конфигурации (рис. 11.13).

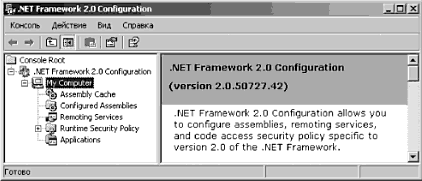


Рис 11.13. Утилита конфигурации .NET Framework 2.0 Configuration

Чтобы построить файл \*.config клиента с помощью этой утилиты, первым шагом должно быть добавление того приложения, которое будет конфигурироваться. Для этого щелчком правой кнопки мыши откройте контекстное меню узла Applications (Приложения) и выберите пункт Add (Добавить). В появившемся диалоговом окне вы *можете*обнаружить приложение для конфигурации при условии, что оно выполнялось ранее с помощью программы Проводник Windows. Если это не так, щелкните на кнопке Other (Другие) и зайдите в папку программы-клиента, которую вы хотите конфигурировать. Для данного примера следует выбрать приложение VbNetCarClient.exe, созданное в этой главе ранее (поищите его в папке Bin). После этого вы должны увидеть новый дочерний узел, как показано на рис. 11.14.

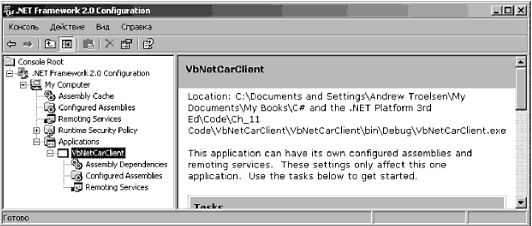


Рис. 11.14. Подготовка к изменению конфигурации VbNetCarClient.exe

Если щелкнуть правой кнопкой мыши на узле VbNetCarClient и активизировать пункт контекстного меню Свойства, то внизу появившегося диалогового окна вы увидите текстовое поле, где можно ввести значения, которые будут приписаны атрибуту privatePath. Просто для проверки введите имя подкаталога TestDir (рис. 11.15).

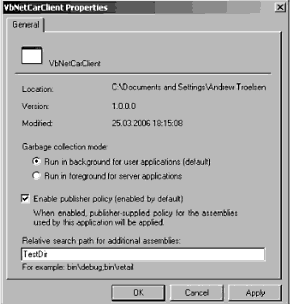


Рис. 11.15. Указание приватного пути зондирования в рамках графического интерфейса

После щелчка на кнопке ОК вы можете посмотреть в каталог VbNetCarClient\ Debug и убедиться, что в типовой файл \*.сonfig (который Visual Studio 2005 создает для программ VB .NET) был добавлен нужный элемент ‹probing›.

**Замечание.**Как вы можете догадаться сами, XML-содержимое, сгенерированное утилитой конфигурации .NET Framework 2.0, можно скопировать в файл App.config Visual Studio 2005 для дальнейшего редактирования. Позволяя инструментам автоматизации генерировать начальное содержимое, вы, очевидно, уменьшаете для себя объем вводимого с клавиатуры текста.

**Общедоступные компоновочные блоки**

Теперь, когда вы понимаете, как инсталлировать и конфигурировать приватные компоновочные блоки, мы с вами можем приступить к рассмотрению роли *общедоступных компоновочных блоков.*Подобно приватному компоновочному блоку, общедоступный компоновочный блок представляет собой набор типов и (возможно) ресурсов. Самой очевидной особенностью общедоступных компоновочных блоков, в отличие от приватных, является то, что одна и та же копия общедоступного компоновочного блока может использоваться разными приложениями.

Так, многие приложения в этой книге указывают ссылку на System.Windows.Forms.dll. Но если заглянуть в каталоги этих приложений, вы *не обнаружите*там приватные копии этого компоновочного блока .NET. Причина в том, что System.Windows.Forms.dll инсталлируется, как общедоступный компоновочный блок. Очевидно, что именно такой подход нужна использовать при создании библиотеки классов, используемой на уровне всей машины.

Общедоступный компоновочный блок не инсталлируется в каталог использующего его приложения. Общедоступные компоновочные блоки устанавливаются в GAC (Global Assembly Cache – глобальный кэш компоновочных блоков). Каталог GAC является подкаталогом с именем assembly в корневом каталоге Windows (например, C:\Windows\assembly), как показано на рис. 11.16.

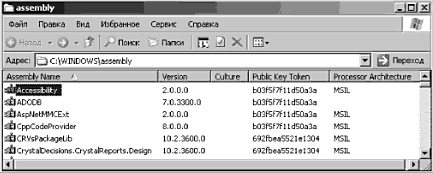


Рис. 11.16. Глобальный кэш компоновочных блоков

**Замечание.**Выполняемые компоновочные блоки (\*.exe) устанавливать в каталог GAC нельзя. Общедоступными компоновочными блоками могут быть только блоки, имеющие вид \*.dll.

**Строгая форма имени**

Перед установкой компоновочного блока в GAC вы должны назначить компоновочному блоку *строгое имя,*которое уникальным образом идентифицирует издателя данного двоичного объекта .NET. При этом следует понимать, что "издателем" может быть и отдельный программист, и подразделение в рамках отдельной компании, и отдельная компания целиком.

В некотором смысле строгое имя является современным .NET-эквивалентом схемы GUID-идентификации COM. Если вы имеете опыт работы с COM, вспомните о том, что AppID (идентификатор приложения) – это GUID (Globally Unique IDentifter – глобальный уникальный идентификатор), характеризующий конкретное COM-приложение. В отличие от GUID-значений в COM (которые являются ничем иным, как 128-разрядными числами), строгие имена создаются на основе двух связанных криптографических ключей (называемых открытым *ключом*и *секретным*ключом). Поэтому строгие имена оказываются гораздо более стойкими в отношении искажений и должны быть ближе к уникальности, чем простые GUID-значения.

Формально строгое имя компонуется из набора связанных данных, в большинстве своем задаваемых следующими атрибутами уровня компоновочного блока.

• Понятное имя компоновочного блока (которое, напоминаем, является именем компоновочного блока без расширения файла)

• Номер версии компоновочного блока (назначаемый с помощью атрибута [AssemblyVersion])

• Значение открытого ключа (назначаемое с помощью атрибута [AssemblyKeyFile])

• Необязательное значение идентификатора культуры, используемого для локализации (назначаемое с помощью атрибута [AssemblyCulture])

• Встроенная *цифровая подпись,*создаваемая с помощью хеширования всего содержимого компоновочного блока с использованием значения секретного ключа

Чтобы создать строгое имя для компоновочного блока, вашим первым шагом должно быть генерирование пары ключей (открытого и секретного) с помощью утилиты sn.exe .NET Framework 2.0 SDK (что мы с вами сделаем чуть позже). Утилита sn.exe генерирует файл, обычно с расширением \*.snk (Strong Name Key – ключ строгого имени), который содержит данные двух разных, но математически связанных ключей (это так называемые "открытый" и "секретный" ключи). Если компилятор C# получит информацию о месте нахождения файла \*.snk, то во время компиляции значение открытого ключа будет записано в манифест создаваемого компоновочного блока с помощью лексемы .publickey.

Компилятор C# также сгенерирует хешированный код на основе всего содержимого компоновочного блока (CIL-кода, метаданных и т.д.). Вы должны знать из главы 3, что *хешированный код*представляет собой числовое значение, уникальным образом характеризующее вводимые данные. Так, при изменении любой части компоновочного блока (даже одного-единственного символа строкового литерала) .NET-компилятор генерирует уже другой хешированный код. Этот хешированный код комбинируется с данными секретного ключа из файла \*. snk для получения цифровой подписи, встраиваемой в CLR-заголовок компоновочного блока. Процесс создания строго именованного компоновочного блока схематически показан на рис. 11.17.

Следует понимать, что данные секретного ключа нигде в манифесте представлены не будут – они используется только при создании цифровой подписи содержимого компоновочного блока (в совокупности с генерируемым хешированным кодом). Напомним, что основной целью применения криптографии на основе открытого и секретного ключей является гарантия того, что во "вселенной" .NET никакая пара компаний, подразделений или индивидуумов не получит одинаковых идентификаторов. Так или иначе, по завершении процесса создания строгого имени компоновочный блок можно будет установить в структуру GAC.

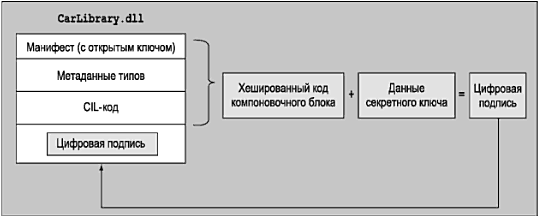


Рис. 11.17. В процессе компиляции на основе открытого и секретного ключей генерируется цифровая подпись, которая затем встраивается в компоновочный блок

**Замечание.**Строгие имена обеспечивают и определенную степень защиты от потенциальных нарушителей, пытающихся модифицировать содержимое компоновочного блока. С учетом этого в рамках .NET считается целесообразным создавать строгое имя для каждого компоновочного блока, а не только для тех компоновочных блоков, которые предназначены для установки в структуру GAC.

**Создание строгого имени для CarLibrary.dll**

Давайте продемонстрируем весь процесс назначения строгого имени компоновочному блоку CarLibrary, созданному в этой главе выше. Откройте соответствующий проект в той среде разработки, которую вы предпочитаете использовать. Первым делом нужно сгенерировать необходимые данные ключей с помощью утилиты sn.exe. Этот инструмент имеет множество опций командной строки, но нам сейчас понадобится только флаг *-*k, который дает команду генерировать новый файл, содержащий информацию открытого и секретного ключей. Создайте новую папку MyTestKeyPair на своем диске C и перейдите в нее в окне командной строки .NET. После этого, чтобы сгенерировать файл MyTestKeyPair.snk, введите следующую команду.

sn -k MyTestKeyPair.snk

Теперь, получив данные своего ключа, сообщите компилятору C# о том, где размещается файл MyTestKeyPair.snk. Обратите внимание на то, что при создании рабочего пространства для любого нового проекта C# в Visual Studio 2005 один из исходных файлов проекта получает имя AssemblyInfo.cs (он размещается в рамках узла Properties в окне Solution Explorer). Этот файл содержит ряд свойств, описывающих компоновочный блок. Атрибут AssemblyKeyFile уровня компоновочного блока можно использовать для информирования компилятора о месте расположения файла \*.snk. Просто укажите путь в виде строкового параметра, например:

[assembly: AssemblyKeyFile (@"C:\MyTestKeyPair\MyTestKeyPair.snk".)]

Поскольку одной из составляющих строгого имени общедоступного компоновочного блока является идентификатор версии, мы укажем его и для CarLibrary.dll. В файле AssemblyInfo.cs найдите другое свойство, имя которого AssemblyVersion. Изначально его значением является 1.0.\*.

[assembly: AssemblyVersion("1.0.\*")]

Напомним, что идентификатор версии .NET компонуется из четырех числовых значений. По умолчанию Visual Studio 2005 автоматически будет выполнять приращение номеров компоновки и варианта (что обозначается групповым символом "\*") при каждой компиляции. Чтобы установить фиксированное значение идентификатора версии для компоновочного блока, замените групповой символ конкретными значениями номера компоновки и варианта.

**// Формат номера версии;**

**// ‹главный›.‹дополнительный›.‹компоновка›.‹вариант›**

**// Для каждого элемента допустимы значения от 0 до 65535.**

[.assembly: AssemblyVersion ("1.0.0.0")]

Теперь компилятор C# имеет всю информацию, необходимую для генерирования данных строгого имени (поскольку вы не указали уникального значения для параметра локализации с помощью атрибута [AssemblyCulture], будут "унаследованы" текущие параметры локализации вашей машины). Выполните компиляцию библиотеки программного кода CarLibrary и с помощью ildasm.exe откройте ее манифест. Вы увидите, что в нем теперь используется лексема .publickey, с помощью которой документируется информация открытого ключа, а с помощью.ver представлен номер версии, указанный атрибутом [AssemblyVersion] (рис. 11.18).

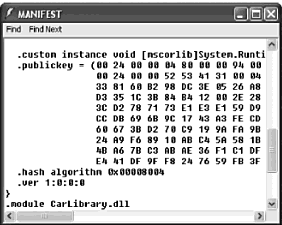


Рис. 11.18. Строго именованный компоновочный блок записывает открытый ключ в манифест

**Назначение строгого имени в Visual Studio 2005**

Перед тем как установить CarLibrary.dll в структуру GAC, заметим, что Visual Studio 2005 позволяет указать место расположения файла \*.snk на странице Properties (Свойства) проекта (в Visual Studio 2005 такой подход оказывается более предпочтительным, поскольку при использовании атрибута [AssemblyKeyFile] генерируется предупреждение компилятора). Выберите вкладку Signing (Подписи) и, указав путь к файлу \*.snk, установите флажок Sign the assembly (Подписать компоновочный блок), как показана на рис. 11.19.

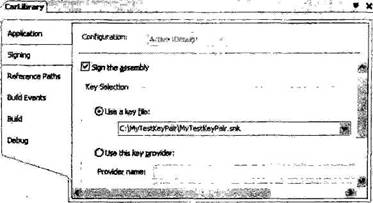


Рис. 11.19. Информация о файле \*.snk на странице свойств проекта

**Установка и удаление общедоступных компоновочных блоков**

Заключительным шагом будет установка (теперь уже строго именованной) библиотеки CarLibrary.dll в структуру GAC. Проще всего установить общедоступный компоновочный блок в структуру GAС, перетащив файл компоновочного блока с помощью мыши в папку C:\Windows\assembly в программе Проводник Windows (это очень удобно при тестировании).

Кроме этого, .NET Framework 2.0 SDK предлагает утилиту командной строки gacutil.exe, которая позволяет просматривать изменять содержимое GAC. В табл. 11.1 показаны некоторые опции gacutil.exe (используйте флаг /?*,*чтобы увидеть все опции),

Таблица 11.1. Опции gacutil.exe

|  |  |
| --- | --- |
| **Опция** | **Описание** |
| /i | Устанавливает строго именованный компоновочный блок в структуру GAC |
| /u | Удаляет компоновочный блок из структуры GAC |
| /l | Отображает компоновочные блоки (или конкретный компоновочный блок) в структуре GAC |

Используя любой из указанных подходов, установите CarLibrary.dll в структуру GAC. После этого вы должны увидеть, что ваша библиотека в структуре присутствует и учитывается (рис. 11.20).

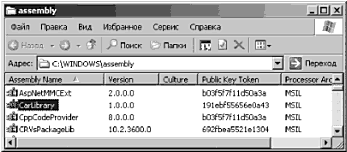


Рис. 11.20. Строго именованная общедоступная библиотека CarLibrary (версия 1.0.0.0)

Замечание. При щелчке правой кнопкой мыши на пиктограмме любого компоновочного блока открывается контекстное меню, с помощью которого можно открыть страницу свойств компоновочного блока или деинсталлировать соответствующую версию (это эквивалентно использования флага /u с утилитой gacutil.exe).

**Отложенная подпись**

При создании своих собственных компоновочных блоков .NET вы можете назначать строгие имена, используя свой персональный файл \*.snk. Но ваша компания или подразделение могут отказать вам в доступе к главному файлу \*. snk. Ввиду исключительной важности файла, содержащего открытый и секретный ключи, этому удивляться не следует, но это, очевидно, является проблемой, поскольку у вас (как у разработчика) будет часто возникать необходимость установки компоновочных блоков в структуру GAC с целью тестирования. Чтобы позволить такое тестирование без предоставления настоящего файла \*.snk, вы можете использовать метод *отложенной подписи.*В случае с файлом CarLibrary.dll в использовании такого подхода нет никакой необходимости, но мы все же предоставим краткое описание соответствующей процедуры.

Процедура отложенной подписи начинается правомочным лицом, имеющим доступ к файлу \*.snk, с извлечения из этого файла значения открытого ключа. Для этого используется sn.exe с опцией -р, позволяющей создать новый файл, содержащий значение открытого ключа.

sn -p myKey.snk testPublicKey.snk

Файл testPublicKey.snk можно предоставить всем разработчикам для создания и проверки строго именованных компоновочных блоков. Чтобы сообщить компилятору C# о том, что соответствующий компоновочный блок должен использовать процедуру отложенной подписи, разработчик должен установить для атрибута AssemblyDelaySign значение true (истина), а также указать файл с псевдоключом, как параметр атрибута AssemblyKeyFile. Ниже показаны строки, которые следует ввести в файл AssemblyInfo.cs проекта.

[assembly: **AssemblyDelaySign**(true)]

[assembly: AssemblyKeyFile(@"C:\MyKey\**testPublicKey.snk**)]

**Замечание.**При использовании Visual Studio 2005 те же атрибуты можно создать "визуально", используя возможности, предлагаемые на странице свойств проекта.

После разрешения отложенной подписи для компоновочного блока следующим шагом является отключение процесса проверки подписи, который для компоновочных блоков, установленных в GAC, выполняется автоматически. Чтобы пропустить процесс проверки подписи на текущей машине, укажите (для sn.exe) опцию -vr.

sn.exe -vr MyAssembly.dll

По завершении тестирования компоновочный блок можно отправить уполномоченному объекту, имеющему доступ к настоящему файлу с открытым и секретным ключами, чтобы обеспечить двоичному файлу настоящую цифровую подпись. Здесь снова необходимое решение обеспечивает sn.exe, на этот раз при использовании флата -r.

sn.exe -r MyAssembly.dll C:\MyKey\myKey.snk

Чтобы в заключение активизировать процесс проверки подписи, применяется флаг -vu.

sn.exe -vu MyAssembly.dll

Конечно, если вы (или ваша компания) создаете компоновочные блоки только дан внутреннего использования, вам процедура отложенной подписи может никогда не понадобиться. Но если вы создаете компоновочные блоки .NET для внешних потребителей, возможность отложенной подписи позволяет обеспечить безопасность с разумными ограничениями для всех заинтересованных сторон.

**Использование общедоступных компоновочных блоков**

При построении приложений, использующих общедоступные компоновочные блоки, единственное отличие от случая использования приватного компоновочного блока заключается в том, как вы ссылаетесь на соответствующую библиотеку в Visual Studio 2005. Фактически с точки зрения используемого инструмента никакой разницы нет (вы все равно используете диалоговое окно Add Reference). Важно понять то, что это диалоговое окно *не позволит* вам сослаться на компоновочный блок путем просмотра папки assembly. Попытки сделать это будут бесполезными – возможность сослаться на выделенный компоновочный блок предоставлена не будет (рис. 11.21).

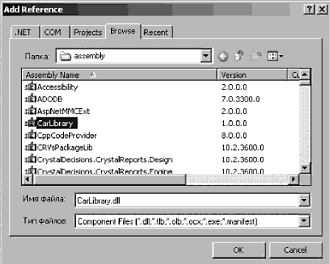


Рис. 11.21. Неправильно! Visual Studio 2005 не позволяет сослаться на общедоступный компоновочный блок путем перехода в папку assembly

Вместо этого на вкладке Browse нужно перейти в каталог \Bin\Debug *оригинального*проекта (рис. 11.22).

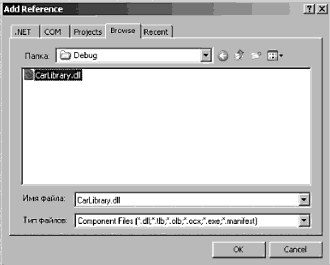


Рис. 11.22. Правильно! В Visual Studio 2005 для ссылки на общедоступный компоновочный блок нужно перейти в каталог \Bin\Debug соответствующего проекта

Учитывая этот (раздражающий) факт, создайте новое консольное приложение C# с именем SharedCarLibClient и проверьте возможность использования своих типов.

using CarLibrary;

namespace.SharedCarLibClient {

 class Program {

 static void Main(string[] args) {

  SportsCar c = new SportsCar();

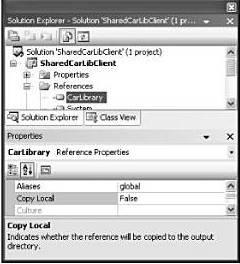
  Console.ReadLine();

  }

 }

}

После компиляции приложения-клиента, в программе Проводник Windows перейдите в каталог, содержащий файл SharedCarLibClient.exe, и убедитесь в том, что Visual Studio 2006 *не скопировала*CarLibrary.dll в каталог приложения-клиента. При ссылке на компоновочный блок, манифест которого содержит значение .publickey, Visual Studio 2005 предполагает, что строго именованный компоновочный блок, вероятнее всего, установлен в структуре GAC и поэтому не "утруждает" себя копированием двоичного файла.



Pиc. 11.23. С помощью свойства Copy Local можно "заставить" систему выполнить копирование строго именованной библиотеки программного кода

В качестве краткого замечания укажем на то, что можно "заставить" Visual Studio 2005 скопировать общедоступный компоновочный блок в каталог клиента. Для этого нужно выбрать компоновочный блок из узла References в окне Solution Explorer, а затем в окне Properties (рис. 11.23) установить для свойства Copy Local (копировать в локальную папку) значение True (истина) вместо значения False (ложь).

**Анализ манифеста SharedCarLibClient**

Напомним, что при генерировании строгого имени компоновочного блока в манифест компоновочного блока записывается значение открытого ключа. Точно так же, когда клиент ссылается на строго именованный компоновочный блок, в его манифест записывается "конденсированное" хешированное значение открытого ключа, обозначенное лексемой .publickey. Если с помощью ildasm.exe открыть манифест SharedCarLibClient.exe, вы увидите там следующее.

.assembly extern CarLibrary {

 .publickeytoken = (21 9E F3 80 C9 34 8A 38)

 .ver 1:0:0:0

}

Если сравнить код открытого ключа, записанный в манифесте клиента со значением для открытого ключа, показанным структурой GAC, вы обнаружите полное совпадение. Напомним, что открытый ключ является одной из составляющих строгого имени, идентифицирующего компоновочный блок. С учетом этого среда CLR загрузит только версию 1.0.0.0 компоновочного блока CarLibrary, открытый ключ которого имеет хешированное значение 219EF380C9348A38. Если среда CLR не найдет компоновочный блок, имеющий такое описание в рамках GAC (и не найдет приватного компоновочного блока с именем CarLibrary в каталоге клиента), то будет сгенерировано исключение FileNotFound (файл не найден).

**Исходный код.**Проект SharedCarLibClient размещен в подкаталога, соответствующем главе 11.

**Конфигурация общедоступных компоновочных блоков**

Подобно приватным компоновочным блокам, открытый компоновочный блок можно конфигурировать с помощью файла \*.config клиента. Конечно, ввиду того, что открытые компоновочные блоки находятся по известному адресу (в структуре GAC), для них не указывается элемент ‹privatePath›, как это делается для приватных компоновочных блоков (хотя, если клиент использует как общедоступные, так и приватные компоновочные блоки, элемент ‹privatePath› в файле \*.config может присутствовать).

Файлы конфигурации приложения можно использовать в совокупности с общедоступными компоновочными блоками для того, чтобы дать указание среде CLR выполнить привязку к *другой*версии конкретного компоновочного блока, т.е. чтобы обойти значение, записанное в манифест клиента. Это может понадобиться по целому ряду причин. Например, представьте себе, что вами была выпущена версия 1.0.0.0 компоновочного блока, а через некоторое время вы обнаружили в ней дефект. Одной из возможностей исправления ситуации может быть перекомпоновка приложения-клиента, чтобы оно ссылалось на новую версию компоновочного блока (скажем, 1.1.0.0). свободную от дефекта, и переустановка обновленного клиента и новой библиотеки на всех соответствующих машинах.

Другим вариантом является поставка новой библиотеки программного кода с файлом \*.config, который автоматически даст среде выполнения инструкцию по привязке к новой версий (свободной от соответствующего дефекта). После установки новой версии библиотеки в структуру GAC оригинальный клиент сможет работать без повторной компиляции и переустановки, а вы не будете опасаться за свою репутацию.

Вот еще один пример. Вы предложили первую версию (1.0.0.0) компоновочного блока, свободного от всяких дефектов, но после месяца-двух его эксплуатации вы добавили в компоновочный блок новые функциональные возможности, позволяющие перейти к версии 2.0.0.0, Очевидно, существующие приложения клиента, которые были скомпилированы в условиях существования только версии 1.0.0.0, не имеют никакого "представления" о новых типах, так что их базовый программный код на эти новые типы не ссылается.

Предполагается, что новые приложения-клиенты будут использовать новые функциональные возможности версии 2.0.0.0. В рамках платформы .NET вы имеете возможность отправить версию 2.0.0.0 на целевые машины, чтобы эта новая версия работала вместе с более "старой" версией 1.0.0.0. Если необходимо, существующие клиенты могут динамически перенаправляться на загрузку версии 2.0.0.0 (чтобы получить доступ к ее новым, более совершенным возможностям) с помощью) файла конфигурации приложения, тоже без повторной компиляции и переустановки приложения-клиента.

**Фиксация версии общедоступного компоновочного блока**

Чтобы показать, как осуществляется динамическая привязка к конкретной версии общедоступного компоновочного блока, откроите программу Проводник Windows в скопируйте текущую версию CarLibrary (1.0.0.0) в другой подкаталог (здесь для него выбрано название "Версия 1") корневой папки проекта, чтобы зафиксировать эту версию (рис. 11.24).

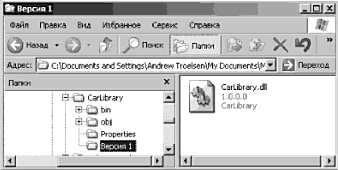


Рис. 11.24. Фиксация текущей версии CarLibrary dll

**Создание общедоступного компоновочного блока версии 2.0.0.0**

Теперь обновите свой проект CarLibrary, добавив в него определение нового перечня MusicMedia, определяющего четыре возможных музыкальных устройства.

**// Содержит информацию об источнике музыки.**

public enum MusicMedia {

 musicCd,

 musicTape,

 musicRadio,

 musicMp3

}

Также добавьте для типа Car новый открытый метод, который позволит вызывающей стороне включить один из имеющихся проигрывателей.

public abstract class Car {

 …

 public void TurnOnRadio(bool musicOn, MusicMedia mm) {

  if (musicOn) MessageBox.Show(string.Format("Шум {0}", mm));

  else MessageBox.Show("И тишина…");

 }

 …

}

Измените конструкторы класса Car, чтобы отображалось окно MessageBox, в котором подтверждается использование CarLibrary именно версии 2.0.0.0.

public abstract class Car {

 …

 public Car() {

  MessageBox.Show("Car 2.0.0.0");

 }

 public Car(string name, short max, short curr) {

  MessageBox.Show("Car 2.0.0.0");

  petName = name; maxSpeed = max; currSpeed = curr;

 }

 …

}

Наконец, до начала новой компиляции не забудьте изменить значение версии этого компоновочного блока на 2.0.0.0 с помощью изменения значения, передаваемого атрибуту [AssemblyVersion].

**// CarLibrary версии 2.0.0.0 (теперь с музыкой!).**

[assembly: Assembly-Version("2.0.0.0"]

Если вы теперь заглянете в папку \Bin\Debug проекта, то увидите, что там присутствует новая версия компоновочного блока (2.0.0.0), в то время как версия 1.0.0.0 в полной безопасности хранится в подкаталоге Версия 1. Установите этот новый компоновочный блок в папку GAC в соответствии с инструкциями, предложенными в этой главе выше. Обратите внимание на то, что теперь вы будете иметь две версии одного и того же компоновочного блока (рис. 11.25).

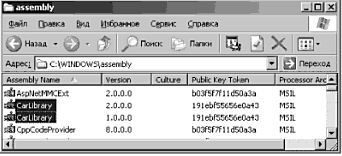


Рис. 11.25. Параллельное выполнение

Если теперь в программе Проводник Windows выполнить имеющуюся программу SharedCarLibClient.exe с помощью двойного щелчка на ее пиктограмме, вы *не увидите*окно с сообщением "Саr 2.0.0.0", поскольку соответствующий манифест специально запрашивает версию 1.0.0.0. Так как же тогда дать указание среде CLR о том, чтобы среда выполнила привязку к версии 2.0.0.0? Я рад, что вы об этом спрашиваете.

**Динамическая привязка к конкретной версии компоновочного блока**

Для того чтобы среда CLR загружала общедоступный компоновочный блок определенной версии, отличной от той версии, которая указана в манифесте компоновочного блока, следует создать файл \*.config с элементом ‹dependentAssembly› внутри. В рамках этого элемента нужно задать элемент ‹assemblyIdentity›, который укажет понятное имя компоновочного блока из соответствующего манифеста клиента (в нашем примере это CarLibrary) и, возможно, необязательное значение атрибута culture (ему можно назначить пустую строку, а можно вообще опустить, если предполагается использовать параметры, предусмотренные для данной машины по умолчанию). Кроме того, в рамках элемента ‹dependentAssembly› следует задать элемент ‹bindingRedirect›, указывающий версию, которая задана в манифесте в *настоящий момент*(атрибут oldVersion), и версию из структуры GAC, которую нужно загружать вместо версии, указанной в манифесте (атрибут newVersion).

В каталоге приложения SharedCarLibClient создайте новый файл конфигурации SharedCarLibClient.exe.config и поместите в него следующие XML-данные. Конечно, значение вашего открытого ключа будет отличаться от того, которое содержится в показанном ниже примере программного кода, но это значение вы можете выяснить путем просмотра манифеста клиента с помощью ildasm.exe или в структуре GAC.

‹configuration›

 ‹runtime›

  ‹assemblyBinding xmlns="urn:schemas-microsoft*-*com:asm.v1"›

   ‹dependentAssembly›

    ‹assemblyIdentity name="CarLibrary" publicKeyToken="191ebf55656e0a43" culture="/›

    ‹bindingRedirect oldVersion= "1.0.0.0" newVersion= "2.0.0.0"/›

   ‹/dependentAssembly›

  ‹/assemblyBinding›

 ‹/runtime›

‹/configuration›

Снова выполните программу SharedCarLibClient.exe. Вы должны увидеть сообщение о том, что загружена версия 2.0.0.0. Если же для атрибута newVersion вы укажете значение 1.0.0.0 (или просто удалите файл \*.config), будет загружена версия 1.0.0.0. поскольку среда CLR найдет в манифесте клиента указание о том, что необходимо использовать версию 1.0.0.0.

В файле конфигурации клиента может присутствовать несколько элементов ‹dependentAssembly›. В нашем случае никакой необходимости в этом нет, но предположим, что манифест SharedCarLibClient.exe ссылается также на общедоступный компоновочный блок MathLibrary версии 2.6.0.0. Если вы захотите перенаправить клиент на использование MathLibrary версии 3.0.0.0 (вдобавок к использованию CarLibrary версии 2.0.0.0), то в этом случае файл SharedCarLibClient.exe.config должен выглядеть так.

‹configuration›

 ‹runtime›

  ‹assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1"›

   ‹dependentAssembly›

    ‹assemblyIdentity name="CarLibrary" publicKeyToken="191ebf55656e0a43" culture="/›

    ‹bindingRedirect oldVersion= "1.0.0.0" newVersion= "2.0.0.0"/›

   ‹/dependentAssembly›

   ‹dependentAssembly›

    ‹assemblyIdentity name="MathLibrary" publicKeyToken="191ebf55656e0a43" culture="/›

    ‹bindingRedirect oldVersion="2.5.0.0" newVersion= "3.0.0.0"/›

   ‹/dependentAssembly›

  ‹/assemblyBinding›

 ‹/runtime›

‹/configuration›

**Снова об утилите конфигурации .NET Framework 2.0**

Вы вправе надеяться, что должна быть какая-то возможность генерирования файлов \*.config общедоступных компоновочных блоков с помощью средств графического интерфейса утилиты .NET Framework 2.0 Configuration. Подобно построению файла \*.сonfig для приватных компоновочных блоков, первый шагом здесь является ссылка на соответствующий файл \*.exe, для которого выполняется конфигурация. Для примера удалите только что созданный вами файл SharedCarLibClient.exe.config. Теперь в окне утилиты .NET Framework 2.0 Configuration добавьте ссылку на SharedCarLibClient.exe, щелкнув правой кнопкой мыши в строке узла Applications (Приложения). Затем раскройте пиктограмму (+) и выберите подузел Configured Assemblies (Сконфигурированные компоновочные блоки). После этого щелкните на ссылке Configure an Assembly (Сконфигурировать компоновочный блок) в правой части окна утилиты.

Вы увидите диалоговое окно, которое позволит вам создать элемент ‹dependentAssembly› с помощью ряда элементов графического интерфейса. Сначала с помощью кнопки переключателя выберите Choose an assembly from the list of assemblies this application uses (Выбрать компоновочный блок из списка компоновочных блоков, используемых данным приложением), что, по сути, означает требование показать манифест. Затем щелкните на кнопке Choose Assembly (Выбрать компоновочный блок).

Появившееся диалоговое окно отобразит не только компоновочные блоки, явно указанные в манифесте клиента, но и компоновочные блоки, на которые указанные компоновочные блоки ссылаются. Для нашего примера выберите CarLibrary. После щелчка на кнопке Finish (Готово) будет показана страница свойств для выбранного объекта манифеста клиента. Там, используя возможности вкладки Binding Policy (Политика привязки ресурсов), вы сможете сгенерировать ‹dependentAssembly›.

На вкладке Binding Policy вы можете установить значения атрибута oldVersion (укажите 1.0.0.0) в текстовом поле Requested Version (Запрошенная версия) и атрибута newVersion (2.0.0.0) текстовом поле New Version (Новая версия). После ввода указанных параметров, вы обнаружите следующий файл конфигурации, сгенерированный для вас системой.

‹?xml version="1.0"?›

‹configuration›

 ‹runtime›

  ‹assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1"›

   ‹dependentAssembly›

    ‹assemblyIdentity name="CarLibrary" publicKeyToken="l91ebf55656e0a43" /›

    ‹publisherPolicy аррlу="yes" /›

    ‹bindingRedirect oldVersion*=*"1.0.0.0" newVersion="2.0.0.0" /›

   ‹/dependentAssembly›

  ‹/assemblyBinding›

 ‹/runtime›

‹/configuration›

**Анализ внутренней структуры GAC**

Итак, все работает. Теперь давайте посмотрим на внутреннюю структуру GAC. При просмотре папки GAG в программе Проводник Windows вы видите ряд пиктограмм, изображающих каждый из общедоступных компоновочных блоков всех имеющихся версий. Эта графическая оболочка обеспечивается COM-сервером shfusion.dll. Но, как вы можете подозревать, за этими пиктограммами должна скрываться сложная (хотя и вполне логичная) структура каталогов.

Чтобы понять, что на самом деле представляет собой структура GAC, откройте окно командной строки и перейдите в каталог assembly.

cd c:\windows\assembly

Выберите в командной строке команду dir. В этом каталоге, среди прочего, вы обнаружите папку с названием GAC\_MISL (рис. 11.26).

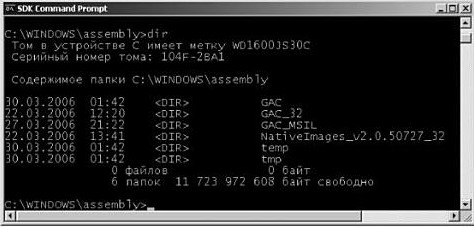


Рис. 11.26. Скрытый подкаталог GAC\_MSIL

Перейдите в каталог GAC\_MSIL и снова выберите команду dir. Теперь вы увидите список подкаталогов, которые имеют в точности такие же имена, как и пиктограммы, отображаемые сервером shfusion.dll. Перейдите в подкаталог CarLibrary и снова выберите команду dir (рис. 11.27).

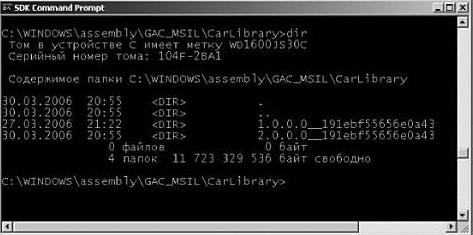


Рис. 11.27. Внутри скрытого подкаталога CarLibrary

Как видите, в структуре GAC для каждой версии общедоступного компоновочного блока создается свой подкаталог, имя которого выбирается по правилу ‹версияКомпоновочногоБлока›\_\_кодОткрытогоКлюча. Если из текущего каталога вы перейдете в каталог CarLibrarу версии 1.0.0.0, то обнаружите там копию соответствующей библиотеки программного кода (рис .11.28).

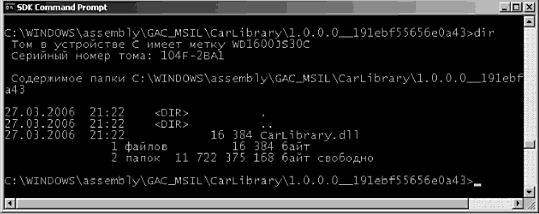


Рис. 11.28. Смотрите! Внутренняя копия GAC библиотеки CarLibrary.dll!

При установке строго именованного компоновочного блока в структуру GAC операционная система расширяет структуру путем создания специального подкаталога в системном каталоге assembly. При таком подходе среда CLR может использовать разные версии, компоновочных блоков, избегая конфликтов, которые иначе могли бы возникать из-за наличия файлов \*.dll с одинаковыми названиями.

**Файлы политики публикации компоновочных блоков**

Следующий вопрос, который мы должны рассмотреть в рамках обсуждения возможностей конфигурации, это роль *политики публикации*компоновочных блоков. Вы только что убедились, что с помощью файлов *\*.*config можно выполнить привязку к конкретной версии общедоступного компоновочного блока в обход версии, указанной в манифесте клиента. Все это просто прекрасно, но представьте себе, что вы являетесь администратором, которому придется переконфигурировать все приложения клиента на данной машине так, чтобы эти приложения использовали компоновочный блок CarLibrary.dll версии 2.0.0.0. Ввиду принятого соглашения для имен файлов конфигурации, вам придется многократно копировать одно и то же XML-содержимое во множество мест (еще и предполагается, что вы должны знать, где находятся все файлы, использующие CarLibrary!). Очевидно, что для администратора это будет просто кошмаром.

Политика публикации позволяет "издателю" данного компоновочного блока (вам, вашему подразделению, вашей компании или другому конкретному поставщику) предложить бинарную версию файла \*.config, которая устанавливается в структуру GAC вместе с новейшей версией соответствующе-то компоновочного блока. Преимущество такого подхода в том, что тогда отпадает необходимость в наличии специальных файлов \*.config в каталогах приложений клиента. Среда CLR читает текущий манифест и пытается найти запрошенную версию в структуре GAC. Но если при этом среда CLR обнаруживает файл политики публикации, читаются встроенные в этот файл XML-данные и выполняется соответствующее перенаправление *на уровне GAC.*

Файлы политики публикации создаются средствами командной строки с помощью .NET-утилиты al.exe (это редактор связей компоновочного блока). Этот инструмент имеет очень много опций, но для построения файла политики публикации потребуются указать только следующие данные:

• информацию о размещении файла \*.config или \*.xml, содержащего инструкции перенаправления;

• имя файла, задающего новые параметры политики публикации;

• информацию о размещении файла \*.snk, используемого для создания подписи файла политики публикации;

• номера версии, назначаемой создаваемому файлу политики публикации.

Чтобы построить файл политики публикации, контролирующий библиотеку CarLibrary.dll, нужно использовать следующую команду.

al /link: CarLibraryPolicy.xml /out:policy.1.0.CarLibrary.dll /keyf: C:\MyKey\myKey.snk /v:1.0.0.0

Здесь XML-содержимое включено в файл с именем CarLibraryPolicy.xml. Имя выходного файла, которое должно иметь формат *policy.‹главный(номер версии)›. ‹дополнителъный(номер версии)›.конфигурируемыйКомпоновочныйБлок),*указывается с помощью флага /out. Обратите также внимание на то, что имя файла, содержащего значения открытого и секретного ключей, тоже должно быть представлено, но с помощью опции /keyf. (Поскольку файлы политики публикации являются общедоступными, они должны быть строго именованными.)

В результате использования al.exe вы получите новый компоновочный блок, который можно разместить в структуре GAC для того, чтобы, не используя отдельные файлы конфигурации для каждого приложения, "заставить" все клиенты использовать CarLibrary.dll версии 2.0.0.0.

**Игнорирование файла политики публикации**

Теперь предположим, что вы (как администратор системы) установили файл политики публикации (и новую, более позднюю версию компоновочного блока) на машине клиента. Как обычно и случается, девять из десяти соответствующих приложений перешли к использованию версии 2.0.0.0 без всяких ошибок. Однако в одном из приложений клиента при доступе к CarLibrary.dll версии 2.0.0.0 возникли проблемы (мы с вами знаем, что создать программное обеспечение, которое будет демонстрировать 100%-ную обратную совместимость, практически невозможно).

В таком случае можно построить файл конфигурации для данного "проблемного" клиента с инструкциями, которые позволят среде CLR *игнорировать*установленные в GAC файлы политики публикации. При этом другие приложения клиента, которые могут использовать новый компоновочный блок .NET, с помощью установленного файла политики публикации будут перенаправлены на новый компоновочный блок. Чтобы отключить политику публикации для отдельного клиента, создайте файл \*.сonfig (с подходящим именем), в котором рамках элемента ‹publisherPolicy› установите для атрибута apply значение no. После этого среда CLR будет загружать компоновочный блок той версии, которая указана в манифесте клиента.

‹configuratоn›

 ‹runtime›

  ‹assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1"›

**‹publisherPolicy apply="no"**/›

*‹*/assemblуВinding›

 ‹/runtime›

‹/configuration›

**Элемент ‹codeBase›**

Файлы конфигурации приложения могут также указать базовый программный код. С помощью элемента ‹codeBase› можно дать инструкцию среде CLR искать зависимые компоновочные блоки в указанных местах (например, в общей сетевой папке или в локальном каталоге вне каталога приложения клиента).

**Замечание**. Если значение, присвоенное в рамках элемента ‹codeBase›, указывает на удаленную машину, компоновочный блок будет загружен по требованию в специальный каталог структуры GAC, имеющий специальное название – *кэш загрузки.*Увидеть содержимое кэша загрузки можно с помощью gacutil.exe, указав при запуске этой утилиты опцию /ldl.

С учетом того, что вы уже знаете об установке компоновочных блоков в GAC, будет ясно, что компоновочные блоки, загружаемые с помощью элемента ‹codeBase›, должны быть строго именованными (в конце концов, как же иначе среда CLR смогла бы установить удаленные компоновочные блоки в структуру GAC?).

**Замечание**. Строго говоря, элемент ‹codeBase› можно использовать и для зондирования компоновочных блоков, которые не являются строго именованными. Однако в таком случае адрес компоновочного блока должен задаваться относительно каталога приложения клиента (в этом отношении данный элемент предлагает более широкие возможности, чем элемент ‹privatePath›).

Создайте консольное приложение с именем СodeBaseСlient, установите для него ссылку на CarLibrary.dll версии 2.0.0.0 и измените исходный файл так.

using CarLibrary;

namespace CodeBaseClient {

 class Program {

  static void Main(string[] args) {

   Console.WriteLine("\*\*\*\*\* Забавы с CodeBase \*\*\*\*\*");

   SportsCar с = new SportsCar();

   Console.WriteLine("Создана спортивная машина.");

   Console.ReadLine();

  }

 }

}

Поскольку библиотека CarLibrary.dll была установлена в структуру GAC, вы уже можете выполнить программу. Но для демонстрации применения элемента ‹codeBase› создайте новую папку на своем диске C (например, папку C:\MyAsms) и поместите в эту папку копию CarLibrary.dll версии 2.0.0.0.

Теперь добавьте в проект CodeBaseClient файл App.config (в соответствии с инструкциями, предложенными в этой главе выше) и добавьте в этот файл следующее XML-содержимое (не забывайте о том, что ваше значение .publickeytoken будет другим, и вы можете выяснить его в структуре GAC).

‹configuration›

 ‹runtime›

  ‹assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1"›

   ‹dependentAssembly›

    ‹assemblyIdentity name="SharedAssembly" publicKeyToken="191ebf55656e0a43." /›

**‹codeBase version="2.0.0.0" href="href="file:///C:\MyAsms\CarLibrary.dll" />**

   </dependentAssembly>

  </assemblyBinding>

 </runtime>

</configuration>

Как видите, элемент <codeBase> вложен в элемент <assemblyIdentity>, ис­пользующий атрибуты name и publicKeyToken для указания понятного имени компоновочного блока и соответствующего кода открытого ключа. Сам элемент <codeBase> указывает версию и (с помощью атрибута href) адрес загружаемого компоновочного блока. Если вы удалите CarLibrary.dll версии 2.0.0.0 из струк­туры GAC, этот клиент все равно будет выполняться успешно, поскольку среда CLR сможет найти внешний компоновочный блок в C:\MyAsms.

Однако если вы удалите каталог MyAsms со своей машины, то клиент работать не сможет. Очевидно, что элементы <codeBase> (если таковые присутствуют) име­ют преимущество по сравнению с проверкой GAC.

**Замечание.**Если размещать компоновочные блоки в случайных местах на машине, велика вероят­ность того, что у вас, в конце концов, возникнет необходимость воссоздания реестра системы (из-за соответствующих проблем DLL), поскольку при перемещении или переименовании па­пок, содержащих выполняемые двоичные файлы приложений, имеющиеся связи будут нару­шаться. В связи с этим используйте <codeBase> с осторожностью.

Элемент <codeBase> может оказаться полезным при ссылках на компоновоч­ные блоки, размещенные в сети на удаленной машине. Предположим, что у нас есть доступ к папке, размещенной по адресу http://www.intertechTraining.com. В таком случае для загрузки удаленного файла \*.dll в кэш загрузки GAC на ло­кальной машине следует изменить элемент <codeBase> так.

<codeBase version="2.0.0.0" href="http://www.IntertechTraining.com/Assemblies/CarLibrary.dll"  />

**Исходный код***.*Проект CodeBaseClient размещен в подкаталоге, соответствующем главе 11.

**Пространство имен System.Configuration**

До этого времени все файлы \*.config, показанные в этой главе, состояли из из­вестных XML-элементов, по которым среда CLR выясняла адреса внешних компо­новочных блоков. Вдобавок кэтим элементам файл конфигурации клиента может содержать и специальные данные приложения, не имеющие никакого отношения к установке связей. С учетом сказанного становится ясно, почему в .NET Framework используется пространство имен, которое позволяет считывать данные файла кон­фигурации клиента программными средствами.

Пространство имен Sуstem.Configuration определяет небольшой набор типов, которые можно использовать для чтения пользовательских установок из файла \*.config клиента. Эти пользовательские установки должны задаваться в контексте элемента <appSettings>. Элемент <appSettings> может содержать произвольное числа элементов <add>, определяющих пары ключей и значений, которые могут извлекаться программными средствами.

Предположим, что у нас есть файл \*.сonfig дата консольного приложения AppConfigReaderApp, в котором определяется строка связи с базой данных и ука­затель на данные timesToSayHello.

<configuration>

**<appSettings**>

  <add key="AppConStr" value="server=localhost;uid='sa';pwd='';database=Cars" />

  <add key="timeToSayHello" value="8" />

 </**appSettings>**

</сonfiguration>

Чтение этих значений для использования приложением клиента осуществляет­ся простым вызовом метода экземпляра GetValue() типа System.Configuration. AppSettingsReader. Как показывает следующий пример программного кода, пер­вый параметр: GetValue() задает имя ключа в файле \*.config, а второй параметр представляет соответствующий тип ключа (получаемый в C# в результате применении операции typeof).

**class Program**{

 static void Main(string[] args) {

**// Создание средства чтения и получение строки соединения.**

  AppSettingsReader ar = new AppSettingsReader();

  Console.WriteLine(ar.GetValue("appConstr", typeof(string)));

**// Получение числа повторений приветствия и выполнение.**

  int numbOfTimes = (int)ar.GetValue("timesToSayHello", typeof(int));

  for (int i = 0; i ‹ numbOfTimes; i++) Console.WriteLine("Йо!");

  Console.ReadLine();

 }

}

Тип класса AppSettingsReader *не задает*способа записи специальных данных приложения в файл \*.config. На первый взгляд это может показаться ограничением, но на самом деле это вполне логично. Сама идея создания файла \*.config заключается в том, чтобы он содержал доступные только для чтения данные, которые должны помочь среде CLR (а также типу AppSettingsReader) правильно установить приложение на соответствующей машине.

**Замечание.**В ходе нашего обсуждения ADO.NET (см. главу 22) вы узнаете об элементе конфигурации ‹connectionStrings› и о других типах пространства имен System.Configuration. Эти элементы, появившиеся в .NET 2.0, предлагают стандартный метод обработки строк соединений.

**Исходный код.**Проект AppConfigReaderApp размещен в подкаталоге, соответствующем главе 11.

**Файл конфигурации машины**

Файлы конфигурации, которые мы с вами рассмотрели в этой главе, имеют одно общее свойство: они относятся к конкретному приложению (вот почему они имеют то же имя, что и соответствующее приложение)*.*Но каждая поддерживающая .NET машина имеет еще и файл, имеющий имя machine.config, который содержит множество параметров конфигурации для управления работой всей платформы .NET (многие из этих параметров не имеют ничего общего с разрешением ссылок на внешние компоновочные блоки).

Платформа .NET использует файл \*.config для каждой своей версии, установленной на локальной машине. Файл machine.config для .NET2.0 можно найти в каталоге C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG (номер вашей версии может быть другим). Открыв указанный файл, вы увидите множество XML-элементов, задающих установки ASP.NET, различные параметры безопасности, поддержку отладки и т.д. Но если вы захотите добавить в файл machine.config (с помощью элемента ‹appSettings›) установки для приложений, применимые в рамках всей машины, вы можете сделать и это.

Этот файл можно редактировать непосредственно, используя программу Блокнот, но следует иметь в виду, что при некорректном изменении этого файла вы можете нарушить работу среды выполнения. Ошибки в этом сценарии могут иметь гораздо более серьезные последствия, чем ошибки в файле \*.config приложения, поскольку ошибки XML в файле конфигурации приложения влияют только на данное приложение, в то время как неправильный XML-код в файле machine.config может вообще блокировать работу конкретной версии .NET.

**Общая схема связей компоновочных блоков**

Теперь, когда мы с вами изучили подробности того, как среда CLR осуществляет поиск запрошенных компоновочных блоков, вспомните о том, что простое на самом деле должно быть простым. Многие (если не все) ваши .NET-приложения будут иметь вид группы приватных компоновочных блоков, размещенных в одном каталоге. В таком случае достаточно просто скопировать соответствующую папку туда, куда требуется, и начать выполнение клиента.

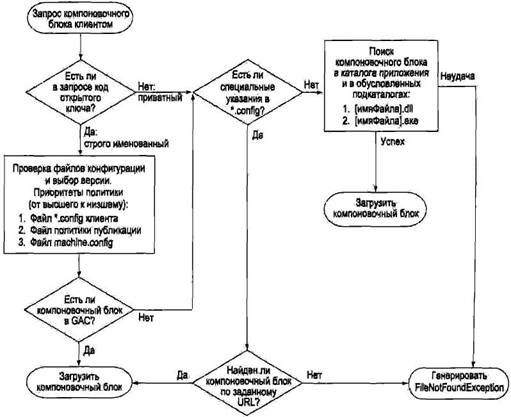


Рис. 11.29. Метод разрешения ссылок компоновочного блока в среде CLR

Однако, вы уже видели, что в процессе разрешения связей среда CLR выполняет проверку на наличие файлов конфигурации клиента и политики публикации компоновочных блоков. В качестве общей схемы пути, который проходит среда CLR при разрешении внешних ссылок компоновочного блока, предлагается схема, показанная на рис. 11.29.

**Резюме**

Эта глава посвящена тому, как среда CLR разрешает ссылки на внешние компоновочные блоки. Глава начинается с рассмотрения содержимого компоновочного блока: заголовка, метаданных, манифеста и CIL-кода. Затем рассматриваются создание одномодульных и многомодульных компоновочных блоков, а также нескольких приложении клиента (на разных языках).

Вы смогли убедиться в том, что компоновочные блоки бывают приватными и общедоступными. Приватные компоновочные блоки копируются в подкаталог клиента. Общедоступные компоновочные блоки устанавливаются в глобальный кэш компоновочных блоков (GAC), и при этом они должны быть строго именованными. Наконец, вы узнали о том, что приватные и общедоступные компоновочные блоки можно конфигурировать, используя XML-файлы конфигурации клиента или, как альтернативу, файл политики публикации.

**ГЛАВА 12. Отображение типов, динамическое связывание и программирование с помощью атрибутов**

Как показано в предыдущей главе, компоновочные блоки являются базовыми элементами установки и среде .NET. С помощью интегрированного обозревателя объектов в Visual Studio 2005 можно рассмотреть открытые типы тех компоновочных блоков, на которые ссылается проект. Внешние средства, такие как ildasm.exe, позволяют увидеть соответствующий CIL-код, метаданные типов и содержимое манифеста компоновочного блока любого бинарного файла .NET, Вдобавок к этим возможностям, доступным во время проектирования компоновочного блока .NET, вы можете получить ту же информацию*программными*средствами, используя объекты пространства имен System.Reflection. В связи с этим мы выясним роль отображения типов и необходимость использования метаданных .NET.

В оставшейся части главы рассматривается ряд тесно связанных вопросов, относящихся к возможностям сервисов отображений. Например, вы узнаете о том, как .NET-клиент может использовать динамическую загрузку и динамическое связывание для активизации типов., о которых у клиента нет полной информации на этапе компиляции. Вы также узнаете, как с помощью системных и пользовательских атрибутов можно добавить в компоновочный блок .NET пользовательские метаданные. Чтобы продемонстрировать перспективы применения этих (да первый взгляд излишне специальных) возможностей, глава завершится примером построения нескольких "встраиваемых" объектов, которые Вы сможете добавить в расширяемое приложение Windows.Form.

**Метаданные типов**

Возможность полного описания типов (классов, интерфейсов, структур, перечней и делегатов) с помощью метаданных является главной особенностью платформы .NET. Многие .NET-технологии, такие как сериализация объектов, удаленное взаимодействие .NET и Web-сервисы XML, требуют, чтобы среда выполнения имела возможность выяснить форматы используемых типов. Возможности межъязыкового взаимодействия, поддержка компилятора и возможности IntelliSense среды разработки тоже зависят от конкретного описания типов.

Важность метаданных очевидна и, возможно, именно поэтому они не являются новой идеей, предложенной в рамках .NET Framework. Технологии Java, CORBA и COM уже использовали аналогичные понятия. Например, для описания типов, содержащихся в серверах COM, используются библиотеки COM-типов (по сути, они представляют собой просто скомпилированный IDL-код). Как и COM, библиотеки программного кода .NET также поддерживают метаданные типов. Конечно, метаданные .NET синтаксически совершенно не похожи на IDL (Interface Definition Language – язык описания интерфейсов, используется в COM-технологиях для спецификации интерфейсов объектов COM). Напомним, что просматривать метаданные типов компоновочного блока позволяет утилита ildasm.exe (см. главу 1), Если вы откроете с помощью ildasm.exe любой компоновочный блок \*.dll или \*.exe, созданный вами в процессе изучения материала этой книги (например, CarLibrary.dll), и нажмете комбинацию клавиш ‹Ctrl+M›, то увидите соответствующие метаданные (рис. 12.1).

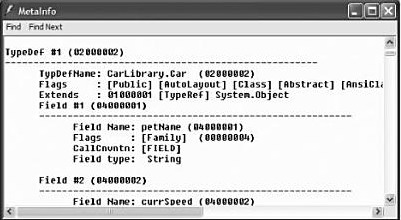


Рис. 12.1. Просмотр метаданных компоновочного блока

Как видите, ildasm.exe отображает метаданные .NET-типа очень подробно (двоичный формат оказывается гораздо более компактным). Если бы здесь потребовалось привести описание метаданных компоновочного блока CarLibrary.dll целиком, оно бы заняло несколько страниц. Это было бы лишней тратой вашего времени (и бумаги тоже), так что давайте рассмотрим метаданные только ключевых типов из компоновочного блока CarLibrary.dll.

**Анализ метаданных перечня EngineState**

Каждый тип, определенный в компоновочном блоке, обозначен маркером "TypeDef #n" (где TypeDef – это сокращение от *type definition,*что в переводе означает *определение типа*). Если описываемый тип использует тип, определённый в рамках другого компоновочного блока .NET, то для ссылки на такой тип используется "TypeRef #n" (где TypeRef – это сокращение от *type reference,* в переводе *ссылка на тип*). Если хотите, TypeRef можно считать указателем на полное определение метаданных соответствующего типа. По существу, метаданные .NET представляют собой множество таблиц, явно описывающих все определения типов (TypeDef) и все типы, на которые имеются ссылки (TypeRef). Все это можно увидеть в окне просмотра метаданных ildasm.exe.

В случае CarLibrary.dll одно из описаний TypeDef в метаданных соответствуeт перечню CarLibrary.EngineState (у вac номер TypeDef может быть другим: нумерация TypeDef соответствует порядку, в котором компилятор C# обрабатывает соответствующие типы).

**TypeDef #1**

**-------------------------------------------------------------**

 TypDefName: **CarLibrary.EngineState**(020000002)

 Flags: [Public] [AutoLayout] [Class] [Sealed] [AnsiClass] (00000101)

 Extends: 01000001 [TypeRef] System.Enum

…

**Field #2**

**-------------------------------------------------------------**

 Field Маше: **engineAlive**(04000002)

 Flags: [Public] [Static] [Literal] [HasDefault] (00008056)

 DefltValue: (I4) 0

 CallCnvntn: [FIELD]

 Field type: ValueClass CarLibrary.EngineState

…

Метка TypDefName используется для имени типа. Метка метаданных Extends используется для указания базового класса данного типа .NET (в данном случае это тип System.Enum, обозначенный как TypeRef). Каждое поле перечня обозначено меткой "Field #n". Для примера здесь представлены только метаданные поля EngineState.engineAlive.

**Анализ метаданных типа Car**

Вот часть дампа типа Car, которая иллюстрирует следующее:

• способ определения полей *в*терминах метаданных .NET;

• представление методов в метаданных .NET;

• отображение свойства типа в пару специальных членов-функций.

**TypeDef #3**

**-------------------------------------------------------------**

 TypDefName: **CarLibrary**.**Car**(02000004)

 Flags: [Public] [AutoLayout] [Class] [Abstract] [AnsiClass] (00100081)

 Extends: 01000002 [TypeRef] System.Object

 Field #1

**-------------------------------------------------------------**

  Field Name: petName (04000008)

  Flags: [Family] (00000004)

  CallCnvntn: [FIELD]

  Field type: String

…

 Method #1

**-------------------------------------------------------------**

  MethodName:.ctor (06000001)

  Flags: [Public] [HideBySig] [ReuseSlot] [SpecialName] [RTSpecialName] [.ctor] (00001886)

  RVA: 0x00002050

  ImplFlags: [IL] [Managed] (00000000)

  CallCnvntn: [DEFAULT]

  hasThis

  ReturnType: Void

  No arguments.

…

 Property #1

**-------------------------------------------------------------**

  Prop.Name: PetName (17000001)

  Flags: [none] (00000000)

  CallCnvntn: [PROPERTY]

  hasThis

  ReturnType: String

  No arguments.

  DefltValue:

  Setter: (06000004) **set\_PetName**

  Getter: (06000003) **get\_PetName**

  0 Others

…

Прежде всего, отметьте то, что метаданные класса Car указывают базовый класс типа и включают различные флаги, использовавшиеся конструктором типа при его создании (такие как [public], [abstract] и т.п.). Методы (например, конструктор класса Car) описаны с учетом их имени, параметров и возвращаемого значения. Наконец, обратите внимание на то, что свойства представляются внутренними методами get\_ /set\_ с использованием меток Setter/Getter метаданных .NET. Как и следует ожидать, производные типы Car (это SportsCar и MiniVan) описываются аналогично.

**Анализ TypeRef**

Напомним, что метаданные компоновочного блока описывают не только множество внутренних типов (Car, EngineState и т.д.), но и внешние типы, на которые ссылается данный компоновочный блок. Например, поскольку CarLibrary.dll Определяет два перечня, в описании присутствует блок TypeRef для типа System.Enum.

**TypeRef #1**(01000001)

**-------------------------------------------------------------**

Token: 0x01000001

ResolutionScope: 0x23000001

TypeRefName: **System.Enum**

 MemberRef #1

**-------------------------------------------------------------**

 Member: (0a00000f) ToString:

 CallCnvntn: [DEFAULT] hasThis

 ReturnType: String

 No arguments.

**Представление метаданных компоновочного блока**

Окно метаданных ildasm.exe позволяет также просмотреть метаданные самого компоновочного блока, для обозначения которых используется метка Assembly. Следующий фрагмент листинга показывает, что информация, представленная в таблице Assembly, аналогична информации, получаемой в окне ildasm.exe через пиктограмму MANIFEST (и это совсем не удивительно). Вот часть манифеста CarLibrary.dll (версии 2.0.0.0).

**Assembly**

**-------------------------------------------------------------**

 Token: 0x20000001

 Name: CarLibrary

 Public Key: 00 24 00 00 04 80 00 00 // и т.д.

 Hash Algorithm: 0x00008004

 Major Version: 0x00000002

 Minor Version: 0x00000000

 Build Number: 0x00000000

 Revision Number: 0x000000000

 Locale: ‹null›

 Flags: [SideBySideCompatible] (00000000)

**Представление ссылок на другие компоновочные блоки**

Вдобавок к метке Assembly и набору меток TypeDef и TypeRef метаданные .NET используют метки "AssemblyRef #n", чтобы обозначить внешние компоновочные блоки. Например, поскольку CarLibrary.dll использует тип MessageBox, в окне метаданных вы обнаружите метку AssemblyRef для System.Windows.Forms.

**AssemblyRef #2**

**-------------------------------------------------------------**

 Token: 0x23000002

 Public Key or Token: b7 7a 5c 56 19 34 e0 89

 Name: System.Windows.Forms

 Version: 2.0.3600.0

 Major Version: 0x00000002

 Minor Version: 0x00000000

 Build Number: 0x00000e10

 Revision Number: 0x00000000

 Locale: ‹null›

 HashValue Blob:

 Flags: [none] (00000000)

**Представление строковых литералов**

В заключение нашего обсуждения метаданных .NET укажем на то, что все строковые литералы базового программного кода представляются в окне метаданных ildasm.exe под знаком метки User Strings, как показано ниже[[1]](http://lib.rus.ec/b/167642/read" \l "n_1" \o "    Символы кириллицы в пользовательских строках программа ildasm.exe просто игнорирует и в окне метаданных не отображает. – Примеч. ред.   ).

User Strings

70000001: (11) L"Car 2.0.0.0"

70000019: (11) L"Jamming {0}"

70000031: (13) L"Quiet time…"

7000004d: (14) L"Ramming speed!"

7000006b: (19) L"Faster is better."

70000093: (16) L"Time to call AAA"

700000b5: (16) L"Your car is dead"

700000d7: (9) L"Be quiet "

700000eb: (2) L"!!"

Пока что не слишком беспокойтесь о точном синтаксисе каждого элемента метаданных .NET. Более важно то, что метаданные .NET дают очень подробное описание всех типов, определенных внутри базового кода, и всех данных, на которые в этом базовом коде имеются ссылки.

Теперь у вас должен возникнуть следующий вопрос: *если вообще нужно что-то знать о метаданных, то как использовать эту информацию в приложениях?*Чтобы получить ответ, давайте рассмотрим такое понятие, как сервисы отображения .NET. А вопрос о пользе предлагаемых ниже подходов мы оставим открытым до рассмотрения соответствующих примеров в конце этой главы. Поэтому наберитесь терпения.

**Замечание.**В окне MetaInfo утилиты ildasm.exe вы обнаружите также ряд меток CustomAttribute, которые используются для обозначения атрибутов, примененных в базовом программном коде. Роль атрибутов .NET мы обсудим в этой главе немного позже.

**Отображение типов в .NET**

В терминах .NET *отображение*обозначает процесс выяснения параметров типа средой выполнения. Используя сервисы отображения, ту же информацию метаданных, которая отображается с помощью ildasm.exe, вы можете получить программно. Например с помощью отображения можно получить список всех типов, содержащихся в данном компоновочном блоке (или в файле \*.netmodule), включая методы, поля, свойства и события, определенные данным типом. Можно также динамически выяснить, какой набор интерфейсов поддерживается данным классом (или структурой), выяснить параметры метода и другие аналогичные подробности (базовые классы, информацию пространства имен, данные манифеста и т.д.).

Подобно любому другому пространству имен, System.Reflection содержит ряд связанных типов. В табл. 12.1 приводится список элементов этого пространства имен, о которых вам следует знать.

**Таблица 12.1.**Некоторые элементы пространства имен System.Reflection

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| Assembly | Этот класс (вместе с множеством связанных типов) предлагает ряд методов, позволяющих загружать, исследовать и обрабатывать компоновочный блок |
| AssemblyName | Класс, позволяющий выяснить многочисленные подробности, касающиеся идентификации компоновочного блока (информацию о версии, параметры локализации и т.д.) |
| EventInfo | Класс, содержащий информацию об указанном событии |
| FieldInfo | Класс, содержащий информацию об указанном поле |
| MemberInfо | Абстрактный базовый класс, определяющий общие характеристики поведения для типов EventInfo, Fieldlnfo, MethodInfo и PropertyInfo |
| MethodInfo | Класс, содержащий информацию об указанном методе |
| Module | Класс, позволяющий получить доступ к указанному модулю многомодульного компоновочного блока |
| ParameterInfo | Класс, содержащий информацию об указанном параметре |
| PropertyInfo | Класс, содержащий информацию об указанном свойстве |

Чтобы понять, как использовать пространство имен System.Reflection для чтения метаданных .NET программными средствами, мы с вами должны сначала ознакомиться с возможностями класса System.Type.

**Класс System.Type**

Класс System.Type определяет ряд членов, которые могут использоваться для чтения метаданных типа, и многие из этих членов возвращают типы из пространства имен System.Reflection. Например, тип Type.GetMethods() возвращает массив типов MethodInfo, тип Type.GetFields() возвращает массив типа FieldInfo и т.д. Полный набор открытых членов System.Type очень велик. В табл. 12.2 предлагается небольшой список наиболее важных из них (подробности описания можно найти в документации .NET Framework 2.0 SDK).

**Таблица 12.2.**Избранные члены System.Type

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| IsAbstract IsArray IsClass IsCOMObject IsEnum IsGenerlcTypeDefinition IsGenericParameter Islnterface IsPrimitive IsNestedPrivate IsNestedPublic IsSealed IsValueType | Эти свойства (наряду с другими аналогичными) позволяют выяснить ряд основных характеристик соответствующего объекта Туре (например, является ли этот объект абстрактным методом, массивом, вложенным классом и т.д.) |
| GetConstructors() GetEvents() GetFields() GetInterfaces() GetMembers() GetMethods() GetNestedTypes() GetProperties() | Эти методы (наряду с другими аналогичными) позволяют получить массив, представляющий все элементы соответствующего вида (интерфейсы, методы, свойства и т.п.). Каждый метод возвращает свой массив (например, GetFields() возвращает массив FieldInfо, GetMethods() возвращает массив MethodInfo и т.д.). Каждый из этих методов имеет также форму единственного числа (GetMethod(), GetProperty() и т.д.), которая позволяет извлечь один конкретный элемент по имени, а не все связанные элементы |
| FindMembers() | Возвращает массив типов MemberInfo на основе заданных критериев поиска |
| GetType() | Статический метод, возвращающий экземпляр Туре по заданному строковому имени |
| InvokeMember() | Позволяет выполнить динамическую привязку к заданному элементу |

**Получение Туре с помощью System.Object.GetType()**

Экземпляр класса Туре можно получить множеством способов. Нельзя только непосредственно создать объект Туре, используя для этого ключевое слово new, поскольку класс Туре является абстрактным. Чтобы привести пример одной из допустимых возможностей, напомним, что System.Object определяет метод GetType(), который возвращает экземпляр класса Туре, представляющий метаданные соответствующего объекта.

**// Получение информации типа с помощью экземпляра SportsCar.**

SportsCar sc = new SportsCar();

Type t = sc.**GetType**();

Очевидно, что этот подход будет оправдан только в том случае, когда вы имеете информацию о соответствующем типе (в данном случае это тип SportsCar) во время компиляции. При этом становится ясно, что такие инструменты, как ildasm.exe, не могут получать информацию о типах путем непосредственно вызова System.Object.GetType(), поскольку ildasm.exe не компилируется вместе с пользовательскими компоновочными блоками.

**Получение Туре с помощью System.Type.GetType()**

Более гибкий подход обеспечивается использованием статического члена GetType() класса System.Type с указанием абсолютного имени соответствующего типа в виде строки. При использовании такого подхода для извлечения метаданных уже *не требуется*информация о типе во время компиляции, поскольку Type.GetType() использует экземпляр "вездесущего" System.String.

Метод Туре.GetType() перегружен, чтобы можно было указать два параметра типа Boolean, один из которых контролирует необходимость генерирования исключения, когда тип не найден, а другой – необходимость игнорирования регистра символов в строке. В качестве примера рассмотрите следующий фрагмент программного кода.

**// Получение информации типа с помощью метода Type.GetType()**

**// (не генерировать исключение, если SportsCar не найден,**

**// и игнорировать регистр символов).**

Type t = Type.GetType(''CarLibrary.SportsCar", false, true);

В этом примере обратите внимание на то, что в строке, которую вы передаете в GetType(), ничего не говорится о компоновочном блоке, содержащем данный тип. В этом случае предполагается, что соответствующий тип определен в рамках компоновочного блока, выполняемого в настоящий момент. Если же вы хотите получить метаданные для типа из внешнего приватного компоновочного блока, то строковый параметр должен иметь формат абсолютного имени типа, за которым через запятую должно следовать понятное имя компоновочного блока, содержащего этот тип.

**// Получение информации типа из внешнего компоновочного блока.**

Type t = null;

t = Type.**GetType**("CarLibrary.SportsCar, CarLibrary");

Следует также знать о том, что в строке, передаваемой методу GetType(), может присутствовать знак "плюс" (+), используемый для обозначения *вложенного типа.*Предположим, что мы хотим получить информацию для перечня (SpyOptions), вложенного в класс JamesBondCar. Для этого мы должны написать следующее.

**// Получение информации типа для вложенного перечня**

**// в рамках имеющегося компоновочного блока.**

Type t = Type.GetType(''CarLibrary. JamesBondCar+SpyOptions");

**Получение Туре с помощью typeof()**

Наконец, можно получить информацию типа с помощью операции C# typeof.

**// Получение Туре с помощью typeof.**

Type t = **typeof(**SportsCar);

Подобно методу Type.GetType(), операция typeof оказывается полезной тем, что при ее использовании нет необходимости сначала создавать экземпляр объекта, чтобы затем извлечь из него информацию типа. Но при этом ваш базовый код все равно должен иметь информацию о типе во время компиляции.

**Создание пользовательского приложения для просмотра метаданных**

Чтобы очертить общие контуры процесса отображения (а также привести пример использования System.Type), мы создадим консольное приложение, которое назовем MyTypeViewer. Эта программа будет отображать подробную информацию о методах, свойствах, полях и поддерживаемых интерфейсах (и другую информацию) для любого типа из MyTypeViewer, а также из mscorlib.dll (напомним, что все приложения .NET автоматически получают доступ к этой базовой библиотеке классов).

**Отображение методов**

Мы модифицируем класс Program, чтобы определить ряд статических методов, каждый из которых будет иметь один параметр System.Type и возвращать void. Начнем с метода ListMethods(), который (как вы можете догадаться сами) печатает имена всех методов, определенных указанным на входе типом. При этом заметим, что Type.GetMethods() возвращает массив типов System.Reflection.MethodInfo.

**// Отображение имен методов типа.**

public static void ListMethods(Type t) {

 Console.WriteLine("\*\*\*\*\* Методы \*\*\*\*\*");

 MethodInfo[] mi = t.GetMethods();

 foreach (MethodInfo m in mi) Console.WriteLine("-›{0}", m.Name);

 Console.WriteLine(");

}

Здесь с помощью свойства MethodInfo.Name просто печатается имя метода. Как и следует предполагать, MethodInfo имеет много других членов, которые позволяют выяснить, является ли метод статическим, виртуальным или абстрактным. Кроме того, тип MethodInfo позволяет получить возвращаемое значение метода и множество его параметров. Реализацию ListMethods() мы с вами проанализируем чуть позже.

**Отображение полей и свойств**

Реализация ListFields() будет аналогичной. Единственным отличием будет вызов Type.GetFields(), а результирующим массивом будет FieldInfo. Для простоты мы печатаем только имена полей.

**// Отображение имен полей типа.**

public static void ListFields(Type t) {

 Console.WriteLine("\*\*\*\*\* Поля \*\*\*\*\*");

 FieldInfo[] fi = t.GetFields();

 foreach (FieldInfo field in fi) Console.WriteLine("-›{0}", field.Name);

 Console.WriteLine(");

}

Логика отображения свойств типа аналогична.

**// Отображение имен свойств типа.**

public static void ListProps(Type t) {

 Console.WriteLine("\*\*\*\*\* Свойства \*\*\*\*\*");

 PropertyInfo[] pi = t.GetProperties();

 foreach(PropertyInfo prop in pi) Console.WriteLine("-›{0}", prop.Name);

 Console.WriteLine(");

}

**Отображение реализованных интерфейсов**

Теперь построим метод ListInterfaces(), который будет печатать имена интерфейсов, поддерживаемых указанным на входе типом. Единственным заслуживающим внимания моментом здесь является вызов GetInterfaces(), возвращающий массив System.Types. Это логично, поскольку интерфейсы тоже являются типами.

**// Отображение реализованных интерфейсов.**

public static void ListInterfaces(Type t) {

 Console.WriteLine("\*\*\*\*\* Интерфейсы \*\*\*\*\*");

 Type[] ifaсes = t.GetInterfaces();

 foreach (Type i in ifaces) Console.WriteLine("-› {0}", i.Name);

}

**Отображение вспомогательной информации**

Наконец, мы рассмотрим еще один вспомогательный метод. который будет отображать различные статистические характеристики типа (является ли тип обобщенным, какой тип для него является базовым, изолирован ли он и т.д.).

**// Отображаются для полноты картины.**

public static void ListVariousStats(Type t) {

 Console.WriteLine("\*\*\*\*\* Вcпомогательная информация \*\*\*\*\*");

 Console.WriteLine("Базовый класс: {0}", t.**BaseType**);

 Console.WriteLine("Это абстрактный тип? {0}", t.**IsAbstract**);

 Console.WriteLine("Это изолированный тип? {'0}", **t.IsSealed**);

 Console.WriteLine("Это обобщенный тип? {0}", t.**IsGenericTypeDefinition**);

 Console.WriteLine("Это тип класса? {0}", t.**IsClass**);

 Console.WriteLine(");

}

**Реализация Main()**

Метод Main() класса Program запрашивает у пользователя абсолютное имя типа. После получения строковых данных они передаются методу Туре.GetType(), а извлеченный объект System.Type отправляется каждому из вспомогательных методов. Это повторяется до тех пор, пока пользователь не нажмет клавишу ‹Q›, чтобы завершить выполнение приложения.

**// Здесь необходимо указать пространство имен отображения.**

using System;

using System.Reflection;

...

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Добро пожаловать в MyTypeViewer! \*\*\*\*\*");

 string typeName = ";

 bool userIsDone = false;

 do {

  Console.WriteLine("\nВведите имя типа");

  Console.Write("или нажмите Q для выхода из приложения: ");

**// Получение имени типа.**

  typeName = Console.ReadLine();

**// Желает ли пользователь завершить работу приложения?**

  if (typeName.ToUpper() = "Q") {

   userIsDone = true;

   break;

  }

**// Попытка отображения типа.**

  try {

   Type t = Type.GetType(typeName);

   Console.WriteLine("");

   ListVariousStats(t);

   ListFields(t);

   ListProps(t);

   ListMethods(t);

   ListInterfaces(t);

  } catch {

   Console.WriteLine("Извините, указанный тип не найден");

  }

 } while (userIsDone);

}

К этому моменту приложение MyTypeViewer.exe уже готово для тестового запуска. Запустите это приложение и введите следующие абсолютные имена (помните о том, что при используемом здесь варианте вызова Туре.GetType() строки имен оказываются чувствительными к регистру символов).

• System.Int32

• System.Collections.ArrayList

• System.Threading.Thread

• System.Void

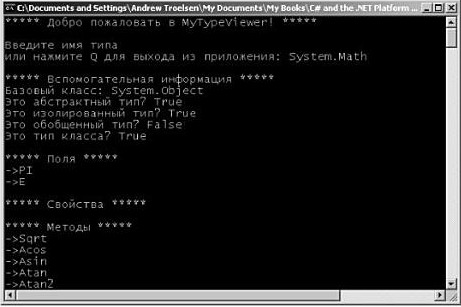
• System.IO.BinaryWriter

• System.Math

• System.Console

• MyTypeViewer.Program

На рис. 12.2 показана информация для случая, соответствующего выбору типа System.Math.



Риc. 12.2. Отображение System.Math

**Отображение параметров и возвращаемых значений методов**

Итак, всё работает. Теперь немного усовершенствуем наше приложение. В частности, модифицируем вспомогательную функцию ListMethods(), чтобы получать не только имя метода, но и возвращаемое значение, а также входные параметры. Для решения именно таких задач тип MethodInfo предлагает свойство ReturnType и метод GetParameters()*.*

В следующем фрагменте программного кода обратите внимание на то, что строка, содержащая тип и имя каждого из параметров, строится с помощью вложенного цикла foreach.

public static void ListMethods(Type t) {

 Console.WriteLine(\*\*\*\*\* Методы \*\*\*\*\*");

 MethodInfo[] mi = t.GetMethods();

 foreach (MethodInfo m in mi) {

**// Получение возвращаемого значения.**

  string retVal = m.ReturnType.FullName;

  string paramInfo = "(";

**// Получение параметров.**

  foreach (ParameterInfo pi in m.GetParameters()) {

   paramInfo += string.Format("{0} {1}", pi.ParameterType, pi.Name);

  }

  paramInfo += ")";

**// Отображение основных характеристик метода.**

  Console.WriteLine("-›{0} {1} (2}", retVal, m.Name, paramInfo);

 }

 Console.WriteLine(");

}

Если выполнить это обновленное приложение теперь, методы соответствующего типа будут описаны более подробно. Для примера на рис. 12.3 показаны метаданные методов для типа System.Globalization.GregorianCalendar.

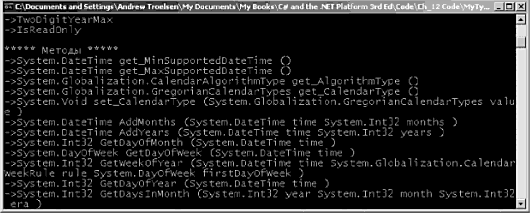


Рис. 12.3. Подробное описание методов System.Globalization.GregorianCalendar

Весьма увлекательно, не так ли? Ясно, что пространство имен System.Reflection и класс System.Type позволяют отображать многие другие характеристики типа, а не только те, которые в настоящий момент реализованы в MyTypeViewer. Вы вправе надеяться на то, что можно будет исследовать события типа, выяснить, какие интерфейсы реализованы явно, получить список обобщенных параметров для заданных членов и проверить множество других характеристик.

Но и в нынешнем своем виде ваш обозреватель объектов уже кое-что умеет. Главным его ограничением, конечно же, является то, что у вас нет никакой возможности отображать объекты, размещенные вне данного компоновочного блока (MyTypeViewer) или всегда доступного mscorlib.dll. В связи с этим остается открытым вопрос: "Как строить приложения, которые могут загружать (и отображать) компоновочные блоки, о которых нет информации во время компиляции?"

**Исходный код**. Проект MyTypeViewer размещен в подкаталоге, соответствующем главе 15.

**Динамически загружаемые компоновочные блоки**

Из предыдущей главы вы узнали о том, как среда CLR использует информацию манифеста компоновочного блока при зондировании компоновочных блоков по внешним ссылкам. Все это, конечно, хорошо, но во многих случаях бывает необходимо "на лету" загрузить компоновочный блок программными средствами, а записей о соответствующем компоновочном блоке в манифесте нет. Формально загрузка внешних компоновочных блоков по запросу называется *динамической загрузкой.*

В рамках System.Reflection определяется класс, имя которого Assembly. Используя этот тип, можно динамически загрузить любой компоновочный блок, а также выяснить его свойства. Используя тип Assembly, можно динамически загружать приватные и общедоступные компоновочные блоки, размещенные в любом месте системы. Класс Assembly предлагает методы (в частности, Load() и LoadFrom()), позволяющие программными средствами получать информацию, аналогичную той, которая содержится в файле \*.config клиента.

Для примера использования динамической загрузки создайте новое консольное приложение с именем ExternalAssemblyReflector. Вашей задачей является построение метода Main(), запрашивающего понятное имя компоновочного блока для динамической загрузки. Ссылка Assembly будет передана вспомогательному методу DisplayTypes(), который просто напечатает имена всех, классов, интерфейсов, структур, перечней и делегатов соответствующего компоновочного блока. Необходимый программный код выглядит довольно просто.

using System;

using System.Reflection;

using System.IO; // **Для определения FileNotFoundException.**

namespace ExternalAssemblyReflector {

 class Program {

  static void DisplayTypesInAsm(Assembly asm) {

   Console.WriteLine("\n\*\*\* Типы компоновочного блока \*\*\*");

   Console.WriteLine("-› {0}", asm.FullName);

   Type[] types = asm.GetTypes();

   foreach (Type t in types) Console.WriteLine("Тип: {0}", t);

   Console.WriteLine(");

  }

  static void Main(string[] args) {

   Console.WriteLine("\*\*\* Обзор внешних компоновочных блоков \*\*\*");

   string asmName = ";

   bool userIsDone = false;

   Assembly asm = null;

   do {

    Console.WriteLine("\nВведите имя компоновочного блока");

    Console.Write("или нажмите Q для выхода из приложения:");

**// Получение имени компоновочного блока.**

    asmName = Console.ReadLine();

**// Желает ли пользователь завершить работу приложения?**

    if (asmName.ToUpper() == "Q") {

     userIsDone = true;

     break;

    }

**// Попытка загрузить компоновочный блок.**

    try {

     asm = Assembly.Load(asmName);

     DisplayTypesInAsm(asm);

    } catch {

     Console.WriteLine("Извините, компоновочный блок не найден.");

    }

   } while (userIsDone);

  }

 }

}

Обратите внимание на то, что статическому методу Assembly.Load() передается только понятное имя компоновочного блока, который вы хотите загрузить в память. Поэтому, чтобы получить отображение CarLibrary.dll с помощью этой программы, перед ее выполнением нужно скопировать двоичный файл CarLibrary.dll в каталог \Bin\Debug приложения ExternalAssemblyReflector. После этого вывод программы будет аналогичен показанному на рис. 12.4.

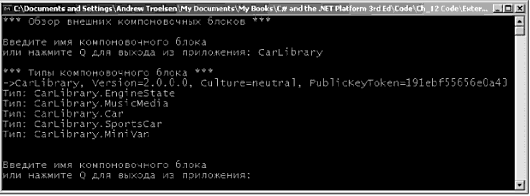


Рис. 12.4. Отображение внешнего компоновочного блока CarLibrary

**Замечание**. Чтобы приложение ExternalAssemblyReflector было более гибким, следует загружать внешний компоновочный блок с помощью A*s*sembsly.LoadFrom(), а не с помощью Assembly.Load(). Тогда вы сможете указать для соответствующего компоновочного блока абсолютный путь (например, C:\MyApp\MyAsm.dll).

**Исходный код.**Проект ExternalAssemblyReflector размещен в подкаталоге, соответствующем главе 12.

**Отображение общедоступных компоновочных блоков**

Как вы можете догадываться, метод Assembly.Load() является перегруженным. Один из вариантов метода Assembly.Load() позволяет указать значение параметра culture (для локализованных компоновочных блоков), а также номер версии и значение открытого ключа (для общедоступных компоновочных блоков).

Весь набор элементов, идентифицирующих компоновочный блок, называют *дисплейным*(или *отображаемым) именем.*По своему формату дисплейное имя представляет собой строку пар имен и значений, разделенных запятыми, которая начинается с понятного имени компоновочного блока и продолжается необязательными определениями (они могут указываться в любом порядке). Вот как выглядит соответствующий шаблон (в скобках указаны необязательные элементы).

**Name (,Culture**= код\_языка) (,**Version**= \_главный\_номер.дополнительный\_номер.номер\_компоновки.номep\_вapиaнтa) (,**PublicKeyToken=**код\_открытого\_ключа)

Значение PublicKeyToken = null в строке, определяющей дисплейное имя, указывает на то, что при связывании проверка строгого имени не требуется и его наличие у компоновочного блока не обязательно. Значений Culture = "" сообщает о необходимости использований значения кода локализации, принятого на машине по умолчанию, например:

**// Загрузка CarLibrary версии 1.0.982.23972 с кодом локализации,**

**// используемым по умолчанию.**

Assembly a = Assembly.Load(@"CarLibrary,Version=1.0.982.23972,PublicKeyToken=null,Culture=''");

Пространство имен System.Reflection предлагает также тип AssemblyName, который позволяет представить указанную выше информационную строку в объектной переменной. Обычно этот класс используется вместе с System.Version, являющимся объектным контейнером для номера версии компоновочного блока. Создав дисплейное имя, вы можете передать его перегруженному методу Assembly.Load().

**// Использование AssemblyName для определения дисплейного имени.**

AssemblyName asmName;

asmName = new AssemblyName();

asmName.Name = "CarLibrary";

Version v = new Version("1.0.982.23972");

asmName.Version = v;

Assembly a = Assembly.Load(asmName);

Чтобы загрузить общедоступный компоновочный блок из GAC, параметр Assembly.Load() должен указать значение publickeytoken. Предположим, на-пример, что вы хотите загрузить компоновочный блок System.Windows.Forms.dll версии 2.0.0.0, предлагаемый библиотеками базовых классов .NET. Поскольку число типов в этом компоновочном блоке очень велико, следующее приложение выводит имена только первых 20 типов.

using System;

using System.Reflection;

using System.IO;

namespace SharedAsmReflector {

  public class SharedAsmReflector {

   private static void DisplayInfo(Assembly a) {

    Console.WriteLine("\*\*\*\*\* Информация о компоновочном блоке \*\*\*\*\*");

    Console.WriteLine("Загружен из GAC? {0}", a.GlobalAssemblyCache);

    Console.WriteLine("Имя: {0}", a.GetName().Name);

    Console.WriteLine("Версия: {0}", a.GetName().Version);

    Console.WriteLine("Культура: {0}", a.GetName().CultureInfo.DisplayName);

    Type[] types = a.GetTypes();

    for (int i = 0; i ‹ 20; i++) Console.WriteLine("Тип: {0}", types[i]);

   }

  }

  static void Main(string[] args) {

   Console.WriteLine("\*\*\*\*\* Отображение общедоступных КБ \*\*\*\*\*\n");

**// Загрузка System.Windows.Forms.dll из GAC.**

   string displayName = null;

   displayName = "System.Windows.Forms," +

    "Version=2.0.0.0," +

    "PublicKeyToken=b77а5c561934e089" +

    @"Culture=''";

   Assembly asm = Assembly.Load(displayName);

   DisplayInfo(asm);

   Console.ReadLine();

  }

 }

}

**Исходный код.**Проект SharedAsmReflector размещен в подкаталоге, соответствующем главе 12.

Чудесно! К этому моменту нашего обсуждения вы должны понять, как использовать некоторые базовые элементы из пространства имен System.Reflection для чтения метаданных компоновочного блока во время выполнения. Здесь я готов признать, что, несмотря на "фактор красоты" предлагаемого подхода, вам по роду своей деятельности вряд ли придется строить пользовательские навигаторы объектов. Но не следует забывать о том, что сервисы отображения являются основой целого ряда других, очень широко используемых подходов в программировании, включая и *динамическое связывание.*

**Динамическое связывание**

Упрощенно говоря, *динамическое связывание,*или *динамическая привязка,*- это подход, с помощью которого можно создавать экземпляры заданного типа и вызывать их члены в среде выполнения и условиях, когда во время компиляции о типе еще ничего не известно. При построении приложения, использующего динамическое связывание в отношении некоторого типа из внешнего компоновочного блока нет смысла устанавливать ссылку на такой компоновочный блок. Поэтому манифест вызывающей стороны и не содержит прямой ссылки.

На данном этапе обсуждения значение динамического связывания может казаться вам непонятным. Если вы имеете возможность использовать "статическую привязку" к типу (например, установить ссылку на компоновочный блок и поместить тип в память, используя ключевое слово C# new), то в этом случае так и нужно сделать. Статическое связывание позволяет обнаружить ошибки уже во время компиляции, а не во время выполнения программы. Однако динамическое связывание оказывается очень важным для построения приложений, обладающих более гибкими возможностями расширения.

**Класс System.Activator**

Класс System.Activator обеспечивает возможность реализации процесса динамической привязки в .NET. Кроме методов, унаследованных от System.Object, сам класс Activator определяет очень небольшое множество членов, многие из которых относятся к средствам удаленного взаимодействия .NET (cм. главу 18). Для нашего примера нам понадобится только метод Activator.CreateInstance(), который используется для создания экземпляра типа в рамках динамической привязки.

Этот метод имеет множество перегруженных вариаций, что обеспечивает ему исключительную гибкость. Самая простая вариация члена CreateInstance() должна получить на вход объект Туре, описывающий элемент, который вы хотите динамически разместить. Создайте новое приложение с именем LateBinding и модифицируйте его метод Main() так, как показано ниже (не забудьте поместить копию CarLibrary.dll в каталог \Bin\Debug проекта).

**// Динамическое создание типа.**

public class Program {

 static void Main(string[] args) {

**// Попытка загрузить локальную копию CarLibrary.**

  Assembly a = null;

  try {

   a = Assembly.Load("CarLibrary");

  } catch(FileNotFoundException e) {

   Console.WriteLine(e.Message);

   Console.ReadLine();

   return;

  }

**// Получение метаданных типа Minivan.**

  Type miniVan = a.GetType("CarLibrary.MiniVan");

**// Динамическое создание Minivan.**

  object obj = Activator**.Createlnstance**(miniVan);

 }

}

Обратите внимание на то, что метод Activator.CreateInstance() возвращает обобщенный System.Object, а не строго типизированный MiniVan. Поэтому если к переменной obj вы примените операцию, обозначаемую точкой, то не увидите никаких членов типа MiniVan. На первый взгляд, можно предположить, что эта проблема решается с помощью явного преобразования типа, но ведь программа не имеет никаких указаний на то, что MiniVan должен иметь при выборе какие-либо преимущества.

Весь смысл динамической привязки заключается в создании экземпляров объектов, для которых нет статической информации. Но как при этом можно вызвать методы объекта MiniVan, сохраненного в переменной System.Object? Ответ: с помощью отображений.

**Вызов методов без параметров**

Предположим, что нам нужно вызвать метод TurboBoost() типа MiniVan. Вы помните, что этот метод приводит двигатель в "абсолютно нерабочее" состояние и генерирует появление информационного блока сообщения. Первым нашим шагом должно быть получение типа MethodInfо для метода TurboBoost() с помощью Type.GetMethod(). Получив MethodInfo, мы сможем вызвать Minivan.TurboBoost() с помощью Invoke(). Для MethodInfo.Invoke() необходимо указать все параметры, которые должны быть переданы методу, представленному с помощью MethodInfo. Эти параметры представляются массивом System.Object (поскольку метод может иметь любое число параметров любого типа).

Наш метод TurboBoost() не имеет параметров, поэтому для него указывается null (в данном случае это и означает отсутствие параметров у метода). Модифицируйте метод Main() так.

static void Main(string[] args) {

**// Попытка загрузить локальную копию CarLibrary.**

 ...

**// Получение типа MiniVan**.

 Type miтiVan = a.GetType("CarLibrary.MiniVan");

**// Динамическое создание MiniVan**.

 object obj = Activator.CreateInstance(miniVan);

**// Получение информации о TurboBoost.**

 MethodInfo mi = miniVan.GetMethod("TurboBoost");

**// Вызов метода ('null' означает отсутствие параметров) .**

 mi.Invoke(obj, null);

}

После этого вы сможете лицезреть сообщение, подобное показанному на рис. 12.5.

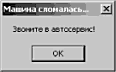


Рис. 12.5. Вызов метода в условиях динамической привязки

**Вызов методов с параметрами**

Чтобы показать пример динамического вызова метода, имеющего параметры, предположим, что тип MiniVan определяет метод, который называется TellChildToBeQuiet().

**// Усмирение вопящих…**

public void TellChildToBeQuiet(**string**kidName, **int**shameIntensity) {

 for (int i = 0; i ‹ shameIntensity; i++)

  MessageBox.Show("Потише, {0}!!", kidName);

}

Метод TellChildToBeQuiet() (приказать ребенку успокоиться) имеет два параметра: строковое представление имени ребенка и целое число, отражающее степень вашего раздражения. При использовании динамического связывания параметры упаковываются в массив объектов System.Object. Для вызова этого нового метода добавьте в свой метод Main() следующий программный код.

**// Динамический вызов метода с параметрами.**

object[] paramArray = new object[2];

paramArray[0] = "Фред"; // Имя ребенка.

paramArray[1] = 4; // Степень досады.

mi = miniVan.GetMethod("TellChildToBeQuiet");

mi.Invoke(obj, paramArray);

Выполнив эту программу, вы сможете увидеть четыре блока сообщений, отражающих намерение пристыдить юного Фреда. Надеюсь, что к этому моменту нашего обсуждения вы уже можете видеть взаимосвязь между отображением, динамической загрузкой и динамическим связыванием. Для вас еще может оставаться неясным ответ на вопрос, *когда*следует использовать указанный подход в приложениях. Завершающий раздел этой главы должен пролить на это свет, но пока что следующей темой нашего рассмотрения будет исследование роли атрибутов .NET.

**Исходный код.**Проект LateBinding размещен в подкаталоге, соответствующем главе 12.

**Программирование с помощью атрибутов**

Как сказано в начале этой главы, одной из задач компилятора .NET является генерирование метаданных для всех определяемых типов и для типов, на которые имеются ссылки. Кроме этих стандартных метаданных, содержащихся в каждом компоновочном блоке, платформа .NET дает программисту возможность встроить в компоновочный блок дополнительные метаданные, используя *атрибуты.*В сущности, атрибуты представляют собой аннотации программного кода, которые могут применяться к заданному типу (классу, интерфейсу, структуре и т.п.), члену (свойству, методу и т.п.), компоновочному блоку или модулю.

Идея аннотирования программного кода с помощью атрибутов не нова. Множество встроенных атрибутов предлагает COM IDL (Interface Definition Language – язык описания интерфейсов), что позволяет разработчику описывать типы COM-сервера. Однако атрибуты COM представляют собой, по сути, лишь набор ключевых слов. Если перед разработчиком COM возникает задача создания пользовательских атрибутов, то эта задача оказывается вполне разрешимой, но ссылаться на такой атрибут в программном коде придется с помощью 128-разрядного номера (GUID), а это, в лучшем случае, слишком обременительно.

В отличие от атрибутов COM IDL (которые, напомним, являются просто ключевыми словами), атрибуты .NET являются типами класса, расширяющими абстрактный базовый класс System.Attribute. При исследовании пространств имен .NET вы можете обнаружить множество встроенных атрибутов, которые можно использовать в приложениях. К тому же вы можете строить свои пользовательские атрибуты, чтобы затем корректировать поведение своих типов с помощью создания новых типов, производных от Attribute.

Следует понимать, что при использовании атрибутов в программном коде вложенные вами метаданные остаются, по сути, бесполезными до тех пор, пока другой фрагмент программного кода не использует явно отображение соответствующей информации. До этого метаданные, вложенные вами в компоновочный блок, просто игнорируются.

**Потребители атрибутов**

Как вы можете догадаться, в комплекте с .NET Framework 2.0 SDK поставляется множество утилит, предназначенных для работы с различными атрибутами. Даже компилятор C# (csc.exe) запрограммирован на проверку определенных атрибутов в процессе компиляции. Например, если компилятор C# обнаруживает атрибут [CLSCompilant], он автоматически проверяет соответствующий элемент на совместимость всех его конструкций с CLS. Если же компилятор C# обнаружит элемент с атрибутом [Obsolete], в окне сообщений об ошибках Visual Studio 2005 появится соответствующее предупреждение.

Вдобавок к инструментам разработки, многие методы из библиотек базовых классов .NET тоже запрограммированы на работу с конкретными атрибутами. Например, если вы хотите сохранить состояние объекта в файле, необходимо указать для класса атрибут [Serializable]. Когда метод Serialize() класса BinaryFormatter обнаруживает указанное свойство, объект автоматически сохраняется в файл в компактном двоичном формате.

Среда CLR также контролирует наличие определенных атрибутов. Возможно, самым известным из атрибутов .NET является [WebMethod]. Если вы хотите открыть метод для запросов HTTP и автоматически кодировать возвращаемое значение метода в формат XML, просто укажите атрибут [WebMethod] для этого метода, и всю рутинную работу среда CLR выполнит сама. Кроме разработки Web-сервисов, атрибуты важны дли системы безопасности .NET, слоя операций удаленного доступа, взаимодействия COM/.NET и т.д.

Наконец, можно строить приложения, которые, наряду с атрибутами библиотек базовых классов .NET. будут отображать пользовательские атрибуты. Такой подход позволяет создавать наборы "ключевых слов", понятных только заданному множеству компоновочных блоков.

**Применение встроенных атрибутов C#**

Как упоминалось выше, библиотека базовых классов .NET предлагает целый ряд атрибутов из разных пространств имен. В табл. 12.3 приводится короткий список некоторых из таких атрибутов (и, конечно же, далеко не всех).

Чтобы привести пример применения атрибутов в C#, предположим, что нам нужно построить класс Motorcycle (мотоцикл), допускающий сериализацию в двоичном формате. Для этого мы должны просто добавить атрибут [Serializable] в определение класса. Если при этом какое-то поле при сериализации сохраняться не должно, то к нему можно применить атрибут [NonSerialized].

**// Этот класс можно сохранить на диске.**

**[Serializable]**

public class Motorcycle {

**// Но это поле сохраняться не должно.**

**[NonSerialized]**

 float weightOfCurrentPassengers;

**// Следующие поля сохраняются.**

 bool hasRadioSystem;

 bool hasHeadSet;

 bool hasSissyBar;

}

**Таблица 12.3.**Малая часть встроенных атрибутов

|  |  |
| --- | --- |
| **Атрибут** | **Описание** |
| [CLSCompliant] | Требует от элемента строгого соответствия правилам CLS (Common Language Specification – общеязыковые спецификации). Напомним, что соответствующие CLS-спецификациям типы гарантированно могут использоваться во всех языках программирования .NET |
| [DllImport] | Позволяет программному коду .NET вызывать библиотеки программного кода C или C++ (которые не являются управляемыми), включая API (Application Programming Interface – программный интерфейс приложения) операционной системы. Заметьте, что [DllImport] не используется при взаимодействии с программным обеспечением COM |
| [Obsolete] | Обозначает устаревший тип или член. При попытке использовать такой элемент программист получит предупреждение компилятора с соответствующим описанием ошибки |
| [Serializable] | Обозначает возможность сериализации класса или структуры |
| [NonSerialized] | Указывает, что данное поле класса или структуры не должно сохраняться в процессе сериализации |
| [WebMethod] | Обозначает доступность метода для вызова через запросы HTTP и требует от среды CLR сохранения возвращаемого значения метода в формате XML (подробности можно найти в главе 25) |

**Замечание.**Указанный атрибут применяется только к элементу, непосредственно следующему за атрибутом. Например, единственным не сохраняемым полем класса Motorcycle будет weightOfCurrentPassengers. Остальные поля при сериализации сохраняются, поскольку весь класс аннотирован атрибутом [Serializable].

Пока что не беспокойтесь о сути самого процесса сериализации объекта (подробности этого процесса будут рассмотрены в главе 17). Обратите внимание только на то, что при использовании атрибута его имя должно заключаться в квадратные скобки.

После компиляции этого класса можете проверить его метаданные с помощью ildasm.exe. Соответствующие атрибуты будут обозначены метками serializable и notserialized (рис. 12.6).

Как вы можете догадаться сами, один элемент может иметь много атрибутов. Предположим, что у нас есть тип класса C# (HorseAndBuggy), обозначенный как serializable, но теперь он считается устаревшим.

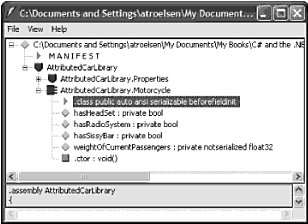


Рис. 12.6. Отображение атрибутов в окне ildasm.exe

Чтобы применить множество атрибутов к одному элементу, используйте список значений, разделенных запятыми.

[Serializable,

Obsolete("Класс устарел, используйте другой транспорт!")]

public class HorseAndBuggy {

 // …

}

В качестве альтернативы, чтобы применить несколько атрибутов к одному элементу, можно просто указать их по порядку (результат будет тем же).

[Serializable]

[Obsolete("Класс устарел, используйте другой транспорт!")]

public class HorseAndBuggy {

 // …

}

**Параметры конструктора для атрибутов**

Мы видим, что атрибут [Obsolete] может принимать нечто похожее на параметр конструктора. Если вы посмотрите на формальное определение атрибута [Obsolete] в окне определения программного кода Visual Studio 2005, то увидите, что данный класс действительно предлагает конструктор, получающий System.String.

public sealed сlass ObsoleteAttribute: System.Attribute {

 public bool IsError { get; }

 public string Message { get; }

 public ObsoleteAttribute(string message, bool error);

**public ObsoleteAttribute(string message)**;

 public ObsoleteAttribute();

}

Когда вы указываете параметры конструктора для атрибута, атрибут не размещается в памяти до тех пор, пока эти параметры не отобразятся другим типом или внешним программным средством. Строки, определенные на уровне атрибута, просто запоминаются в компоновочном блоке, как часть метаданных.

**Атрибут Obsolete в действии**

Теперь, когда класс HorseAndBuggy обозначен как устаревший, при размещении экземпляра этого типа вы должны увидеть соответствующую строку в сообщении, появившемся в окне со списком ошибок Visual Studio 2005 (рис. 12.7).

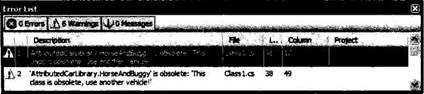


Рис. 12.7. Атрибуты в действии

В данном случае "другим фрагментом программного обеспечения", отображающим атрибут [Obsolete], является компилятор C#.

**Сокращенное представление атрибутов в C#**

При внимательном изучении материала этой главы вы могли заметить, что фактическим именем класса атрибута [Obsolete] является не Obsolete, a ObsoleteAttribute. По соглашению для имен все атрибуты .NET (и пользовательские атрибуты в том числе) должны в конце имени получить суффикс Attribute. Однако, чтобы упростить процедуру применения атрибутов, в языке C# не требуется, чтобы вы обязательно добавляли этот суффикс. Поэтому следующий вариант определения типа HorseAndBuggy будет идентичен предыдущему (при этом только потребуется ввести немного больше символов).

[SerializableAttribute]

[ObsoleteAttribute("Класс устарел, используйте другой транспорт!")]

public class HorseAndBuggy {

 //…

}

Это упрощение предлагается самим языком C#, и следует подчеркнуть, что эту особенность поддерживают не все языки .NET. Так или иначе, к этому моменту нашего обсуждения вы должны понимать следующие основные особенности, касающиеся атрибутов .NET.

• Атрибуты являются классами, производными от System.Attribute.

• Информация атрибутов добавляется в метаданные.

• Атрибуты будут бесполезны до тех пор, пока другой агент не отобразит их.

• Атрибуты в C# применяются с использованием квадратных скобок.

Теперь мы рассмотрим то, как можно строить свои собственные пользовательские атрибуты и пользовательские программы, отображающие встроенные метаданные.

**Создание пользовательских атрибутов**

Первым шагом процесса построения пользовательского атрибута является создание нового класса, производного от System.Attribute. В продолжение автомобильной темы, используемой в этой книге, мы создадим новую библиотеку классов C# с именем AttributedCarLibrary. Соответствующий компоновочный блок определит группу транспортных средств (определения некоторых из них, мы уже увидели выше), и при их описании будет использован пользовательский атрибут VehiсleDescriptionAttribute,

**// Пользовательский атрибут.**

public sealed class VehicleDescriptionAttribute: System.Attribute {

 private string msgData;

 public VehicleDescriptionAttribute(string description) { msgData = description; }

 public VehicleDescriptionAttribute() {}

 public string Description {

  get { return msgData; }

  set { msgData = value; }

 }

}

Как видите, VehicleDescriptionAttribute поддерживает приватную внутреннюю строку (msgData), значение которой можно установить с помощью пользовательского конструктора, а изменять - с помощью свойства типа (Description). Кроме того, что этот класс является производным от System.Attribute, его определение ничем особенным больше не отличается,

**Замечание.**С точки зрения безопасности рекомендуется, чтобы все пользовательские атрибуты…NET создавались, как изолированные классы.

**Применение пользовательских атрибутов**

После получения VehicleDescriptionAttribute из System.Attribute вы можете снабжать свои транспортные средства такими аннотациями, какими пожелаете.

**// Назначение описания с помощью 'именованного свойства'.**

[Serializable,

**VehicleDescription**(Description = "Мой сияющий Харлей")]

public class Motorcycle {

 //…

}

[SerializableAttribute]

[ObsoleteAttribute("Класс устарел, используйте другой транспорт!"), **VehicleDescription**("Старая серая кляча, она уже совсем не та…")]

public class HorseAndBuggy {

 //…

}

[VehicleDescription("Большое, тяжелое, но высокотехнологичное авто"

public class Winnebago {

 //…

}

Обратите внимание на то, что описание класса Motorcycle здесь указано с помощью нового элемента синтаксиса, называемого *именованным свойством.*В конструкторе первого атрибута [VehicleDescription] соответствующее значение System.String устанавливается с помощью пары "имя-значение". При отображении этого атрибута внешним агентом соответствующее значение передается свойству Description (синтаксис именованного свойства здесь корректен только в том случае, когда атрибут предлагает перезаписываемое свойство .NET). В противоположность этому типы HorseAndBuggy и Winnebago не используют синтаксис именованного свойства, а просто передают строковые данные в пользовательский конструктор.

После компиляции компоновочного блока AttributedCarLibrary можно использовать ildasm.exe, чтобы увидеть метаданные с описанием добавленного типа. Так, на рис. 12.8 показано встроенное описание типа Winnebago.

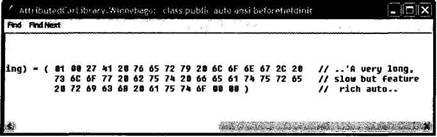


Рис. 12.8. Встроенные данные описания транспортного средства

**Ограничение использования атрибута**

По умолчанию пользовательские атрибуты могут применяться к любой части программного кода (к методам, классам, свойствам и т.д.). Поэтому, если только это имеет смысл, можно использовать VehicleDescription для определения (среди прочего) методов, свойств или полей.

[VehicleDescription("Большое, тяжелое, но высокотехнологичное авто")]

public class Winnebago {

 [VehicleDescription("Мой мощный CD-плейер")]

 public void PlayMusic(bool On) {

  …

 }

}

В некоторых случаях это оказывается именно тем, что нужно. Но в других случаях бывает нужно создать пользовательский атрибут, который должен применяться только к определенным элементам программного кода. Если вы хотите ограничить контекст применения пользовательского атрибута, то при определении пользовательского атрибута нужна применить атрибут [AttributeUsage]. Атрибут [AttributeUsage] позволяет указать любую комбинацию значений (связанных операцией OR) из перечня AttributeTargets.

**// Этот перечень задает возможные целевые значения для атрибута.**

public enum AttributeTargets {

 All, Assembly, Class, Constructor,

 Delegate, Enum, Event, Field,

 Interface, Method, Module, Parameter,

 Property, ReturnValue, Struct

}

Кроме того, [AttributeUsage] позволяет опционально установить именованное свойство (AllowMultiple), которое указывает, может ли атрибут примениться к одному и тому же элементу многократно. Точно так же с помощью именованного свойства Inherited атрибут [AttributeUsage] позволяет указать, должен ли создаваемый атрибут наследоваться производными классами.

Чтобы атрибут [VehicleDescription] мог применяться к классу или структуре только один раз (и соответствующее значение не наследовалось производными типами), можно изменить определение VehicleDescriptionAttribute так.

**// На этот раз для аннотации нашего пользовательского атрибута**

**// мы используем атрибут AttributeUsage.**

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct*,*AllowMultiple = false, Inherited = false)]

public class VehicleDescriptionAttribute: System.Attribute {

 …

}

Теперь если разработчик попытается применить атрибут [VehicleDescription] к чему-либо, кроме класса или структуры, будет сгенерировано сообщение об ошибке компиляции.

**Совет**. Вашей привычкой должно стать явное указание флагов применения для любого создаваемого вами пользовательского атрибута, поскольку не все языки программирования .NET приветствуют использование атрибутов, не имеющих квалификационных указаний!

**Атрибуты уровня компоновочного блока (и уровня модуля)**

Можно также задать применение атрибутов ко всем типам в рамках данного модуля или всех модулей в рамках данного компоновочного блока, если, соответственно, использовать признаки [module:] или [assembly:]. Предположим, что нам нужно гарантировать, чтобы каждый открытый тип, определенный в нашем компоновочном блоке, был CLS-допустимым. Для этого в любой из файлов исходного кода C# нужно добавить следующую строку (заметьте, что атрибуты уровня компоновочного блока должны быть указаны за пределами контекста определения пространства имен).

**// Требование CLS-совместимости для всех открытых типов**

**// в данном компоновочном блоке.**

[assembly:System.CLSCompliantAttribute(true)]

Если теперь добавить фрагмент программного кода, который выходит за пределы спецификации CLS (например, элемент данных без знака)

**// Типы ulong не согласуется с CLS.**

public class Winnebago {

 public **ulong**notCompliant;

}

то будет сгенерирована ошибка компиляции.

**Файл AssemblyInfo.cs в Visual Studio 2005**

По умолчанию Visual Studio 2005 генерирует файл с именем AssemblyInfo.cs (рис. 12.9).

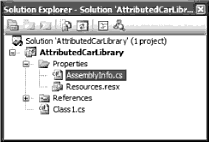


Рис. 12.9. Файл AssemblyInfo.cs

Этот файл является удобным местом для хранения атрибутов, которые должны применяться на уровне компоновочного блока. В табл. 12.4 приводится список некоторых атрибутов уровня компоновочного блока, о которых вам следует знать.

**Исходный код.**Проект AttributedCarLibrary размещен в подкаталоге, соответствующем главе 12.

**Таблица 12.4.**Некоторые атрибуты уровня компоновочного блока

|  |  |
| --- | --- |
| **Атрибут** | **Описание** |
| AssemblyCompanyAttribute | Содержит общую информацию о компании |
| AssemblyCopyrightAttribute | Содержит информацию об авторских правах на продукт или компоновочный блок |
| AssemblyCultureAttribute | Дает информацию о параметрах локализации или языках, поддерживаемых компоновочным блоком |
| AssemblyDescriptionAttribute | Содержит описание продукта или модулей, из которых состоит компоновочный блок |
| AssemblyKeyFileAttribute | Указывает имя файла, содержащего пару ключей, используемых для создания подписи компоновочного блока |
| AssemblyOperatingSystemAttribute | Обеспечивает информацию о том, на поддержку какой операционной системы рассчитан компоновочный блок |
| AssemblyProcessorAttribute | Обеспечивает информацию о том, на поддержку какого процессора рассчитан компоновочный блок |
| AssemblyProductAttribute | Обеспечивает информацию о продукте |
| AssemblyTrademarkAttribute | Обеcпечивает информацию о торговой марке |
| AssemblyVersionAttribute | Указывает информацию версии компоновочного блока, в формате‹*главный.дополнительный.компоновка:вариант*› |

**Отображение атрибутов при статическом связывании**

Как уже упоминалось в этой главе, атрибуты будут бесполезны до тех пор, пока некоторый фрагмент программного обеспечения не выполнит их отображение. После выявления атрибута соответствующий фрагмент программного обеспечения может выбрать подходящее действие. Как и само приложение, этот "фрагмент программного обеспечения" может для выявления пользовательского атрибута использовать статическое или динамическое связывание. Для статического связывания требуется, чтобы приложение-клиент к моменту компиляции уже имели определение соответствующего атрибута (в нашем случае это атрибут VehicleDescriptionAttribute). Компоновочный блок AttributedCarLibrary определяет пользовательский атрибут, как открытый класс, поэтому статическое связывание в данном случае будет наилучшим выбором.

Для иллюстрации процесса отображения пользовательских атрибутов создайте новое консольное приложение C# с именем VehicleDescriptionAttributeReader. Затем установите в нем ссылку на компоновочный блок AttributedCarLibrary. Наконец, поместите в исходный файл \*.cs следующий программный код.

**// Отображение пользовательских атрибутов при статическом связывании.**

using System;

using AttributedCarLibrary;

public class Program {

 static void Main(string [] args) {

**// Получение Type для представления Winnebago.**

  Type t = typeof(Winnebago);

**// Получение атрибутов Winnebago.**

  object[] customAtts = t.GetCustomAttributes(false);

**// Печать описания.**

  Console.WriteLine("\*\*\* Значение VehicleDescriptionAttribute \*\*\*\n");

  foreach(VehicleDescriptionAtttibute v in customAtts) Console.WriteLine("-› {0}\n", v.Description);

  Console.ReadLine();

 }

}

Как следует из самого названия, метод Type.GetCustomAttributes() возвращает массив объектов, представляющих все атрибуты, примененные тому к члену, который представлен с помощью Туре (логический параметр этого метода указывает, следует ли расширить поиск на всю цепочку наследования). После получения списка атрибутов выполняется цикл по всем классам VehicleDescriptionAttribute с выводом на печать значения, полученного свойством Description.

**Исходный код.**Проект VehicleDescriptionAttributeReader размещен в подкаталоге, соответствующем главе 12.

**Отображение атрибутов при динамическом связывании**

В предыдущем примере использовалось статическое связывание и печатались данные описания для типа Winnebago. Это было возможно благодаря тому, что тип класса VehicleDescriptionAttribute был определен, как открытый член компоновочного блока AttributedCarLibrary. Но для отображения атрибутов можно также использовать динамическую загрузку и динамическое связывание.

Создайте новый консольный проект (VehicleDescriptionAttributeReaderLateBinding) и скопируйте AttributedCarLibrary.dll в каталог \Bin\Debug этого проекта. Затем обновите метод Main() так, как предлагается ниже.

using System.Reflection;

 namespace VehicleDescriptionAttributeReaderLateBinding {

  class Program {

   static void Main(string[] args) {

    Console.WriteLine("\*\*\* Описания транспортных средств \*\*\*\n");

**// Загрузка локальной копии AttributedCarLibrагу.**

    Assembly asm = Assembly.Load(AttributedCarLibrary");

**// Получение информации типа для VehicleDescriptionAttribute**.

    Type vehicleDesc = asm.GetType("AttributedCarLibrary.VehicleDescriptionAttribute");

**// Получение информации типа для свойства Description.**

    PropertyInfо propDesc = vehicleDesc.GetProperty("Description");

**// Получение всех типов данного компоновочного блока.**

    Туре[] types = asm.GetTypes()*;*

**// Получение VehicleDescriptionAttribute для каждого типа.**

    foreach (Type t in types) {

     object[] objs = t.GetCustomAttributes(vehicleDesc, false);

**// Итерации по VehicleDescriptionAttribute и печать**

**// описаний с динамическим связыванием.**

     foreach(object о in objs) {

      Console.WriteLine("-› {0}: {1}\n", t.Name, propDesc.GetValue(o, null));

     }

    }

    Console.ReadLine();

   }

  }

}

Если вы внимательно анализировали все примеры этой главы, то листинг этого метода Main() должен быть для вас (более или менее) понятным. Единственным заслуживающим внимания моментом здесь является использование метода PropertyInfo.GetValue() для доступа к свойству. На рис. 12.10 показан соответствующий вывод.

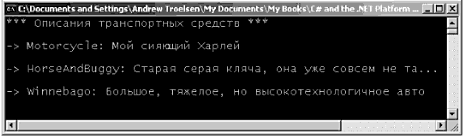


Рис. 12.10. Отображение атрибутов при динамическом связывании

**Исходный код**. Проект VehiсleDescriptionAttributeReaderLateBinding размещен в подкаталоге, соответствующем главе 12.

**Перспективы отображения, статического и динамического связывания и пользовательских атрибутов**

Даже после множества примеров применения соответствующих подходов вам может быть не ясно, когда же следует использовать отображение, динамическую загрузку, динамическое связывание и пользовательские атрибуты в программах. Строго говоря, эти вопросы (которые увлекательны сами по себе) можно отнести, скорее, к теоретической стороне программирования (что можно считать как достоинством, так и недостатком, в зависимости от точки зрения). Чтобы спроецировать эти вопросы на реальность, нам нужен реальный пример. Представьте себе на минуту, что вы работаете в команде программистов, созданной для разработки приложения, удовлетворяющего следующему требованию:

• продукт должен допускать расширение путем подключения дополнительных программных средств, разработанных сторонними производителями.

Но что именно подразумевается под требованием допускать *расширение?*Рассмотрим Visual Studio 2005. При разработке этого приложения в нем были предусмотрены различные "гнезда" для подключения к IDE пользовательских модулей других производителей программного обеспечения. Ясно, что команда разработчиков Visual Studio 2005 не имела при этом возможности установить ссылки на внешние компоновочные блоки .NET, которые эта команда не разрабатывала (так что о статическом связывании не могло быть и речи). Поэтому закономерен вопрос: как именно это приложение смогло предложить необходимые гнезда подключения?

• Во-первых, расширяемое приложение должно предлагать некоторый входной механизм, который мог бы позволить пользователю указать модуль для подключения (это может быть, например, диалоговое окно или опция командной строки). При этом требуется *динамическая загрузка.*

• Во-вторых, расширяемое приложение должно выяснить, обладает ли модуль подходящими функциональными возможностями (например, набором необходимых интерфейсов), чтобы этот модуль можно было добавить в окружение. При этом требуется *отображение.*

• Наконец, расширяемое приложение должно получить ссылку на требуемую инфраструктуру (например, типы интерфейса) и вызвать члены, необходимые для запуска соответствующих функций. При этом часто требуется *динамическое связывание*.

Простыми словами, если расширяемое приложение было запрограммировано с учетом возможности запроса конкретных интерфейсов, оно сможет в среде выполнения проверить, активизирован ли соответствующий тип. Если такая проверка завершится успешно, запрошенный тип сможет поддерживать дополнительные интерфейсы, обеспечивающие полиморфную структуру функциональных возможностей. Именно этот подход был применен командой Visual Studio 2005, и этот подход оказывается не столь сложным для использования, как вы можете ожидать.

**Создание расширяемого приложения**

В следующих разделах мы с вами проанализируем пример, иллюстрирующий процесс создания расширяемого приложения Windows.Forms, которое можно будет расширять за счет функциональных возможностей внешних компоновочных блоков. Сам процесс программирования приложений Windows Forms мы здесь обсуждать не будем (этому будут посвящены главы 19, 20 и 21). Если вы не имеете опыта создания приложений Windows Forms, просто возьмите предлагаемый ниже программный код в качестве образца и следуйте соответствующим инструкциям (или постройте соответствующую консольную альтернативу).

Мы предполагаем, что наше расширяемое приложение должно иметь следующее компоновочные блоки.

• CommonSnappableTypes.dll. Компоновочный блок, содержащий определения типов, которые должны реализовываться каждым подключаемым расширением, поскольку на них будет ссылаться расширяемое приложение Windows Forms.

• CSharpSnapIn.dll. Расширение, созданное на языке C# и использующее типы CommonSnappableTypes.dll.

• VbNetSnapIn.dll. Расширение, созданное на языке Visual Basic .NET и использующее типы CommonSnappableTypes.dll.

• MyPluggableApp.exe. Приложение Windows Forms, функциональные возможности которого можно расширять с помощью подключаемых модулей. Это приложение будет использовать динамическую загрузку, отображение и динамическое связывание для динамического выяснения функциональных возможностей компоновочных блоков, о которых ранее не было ничего известно.

**Создание CommonSnappableTypes.dll**

Первой нашей задачей является создание компоновочного блока, содержащего типы, которые должен использовать каждый подключаемый модуль, чтобы обеспечить возможность его подключения к нашему приложению Windows Forms. Проект библиотеки классов CommonSnappableTypes определяет следующие два типа.

namespace CommonSnappableTypes {

 public interface IAppFunctionality {

  void DoIt();

 }

 [AttributeUsage(AttribyteTargets.Class)]

 public sealed class CompanyInfoAttribute: System.Attribute

  private string companyName;

  private string companyUrl;

  public CompanyInfoAttribute(){}

  public string Name {

   get { return companyName; }

   set { companyName = value; }

  }

  public string Url {

   get { return companyUrl; }

   set { companyUrl = value; }

  }

 }

}

Интерфейс IAppFunctionality обеспечивает полиморфные возможности для всех подключаемых модулей, которые может принять наше расширяемое приложение Windows Forms. Наш пример является исключительно иллюстративным, поэтому здесь интерфейс предлагает единственный метод, DoIt(). В реальности это может быть интерфейс (или набор интерфейсов), позволяющий подключаемому объекту сгенерировать программный код сценария, поместить пиктограмму в окно инструментов или интегрироваться в главное меню приложения.

Тип CompanyInfoAttribute является пользовательским атрибутом, который будет применяться к любому типу класса, размещаемому в контейнере. Исходя из определения этого класса, можно утверждать, что [CompanyInfo] позволяет разработчику расширения сообщить информацию о происхождении подключаемого компонента.

**Создание подключаемого компонента в C#**

Теперь нужно создать тип, реализующий интерфейс IAppFunctionality. Снова, чтобы сосредоточиться на процессе создания расширяемого приложения, здесь предполагается создание самого простого типа. Мы построим библиотеку программного кода C# с именем CSharpSnapIn, которая определит тип класса с именем CSharpModule. Поскольку этот класс должен использовать типы, определенные в CommonSnappableTypes, нам придется установить ссылку на соответствующий двоичный файл (а также на System.Windows.Forms.dll, чтобы выводить необходимые сообщения). С учетом сказанного предлагается использовать следующий программный код.

using System;

using CommonSnappableTypes;

using System.Windows.Forms;

namespace CSharpSnapIn {

 [CompanyInfo(Name = "Intertech Training",

  Url = www.intertechtraining.com)]

 public class TheCSharpModule: IAppFunctionality {

  void IAppFunctionality.DoIt() {

   MessageBox.Show("Вы только что подключили блок C#!");

  }

 }

}

Обратите внимание на то, что здесь используется явная реализация интерфейса IAppFunctionality. Это не обязательно, но идея в том, что единственной частью системы, которой понадобится непосредственное взаимодействие с этим. типом интерфейса, является наше расширяемое приложение Windows.

**Создание подключаемого компонента в Visual Basic .NET**

Теперь, чтобы имитировать стороннего производителя, предпочитающего использовать не C#, a Visual Basic .NET, создадим в Visual Basic .NET новую библиотеку программного кода (VbNetSnapIn), которая будет ссылаться на те же внешние компоновочные блоки, что и CSharpSnapIn. Программный код (снова) будет пред-намерено очень простым.

Imports System.Windows.Forms

Imports CommonSnappableTypes

‹CompanyInfo(Name:="Chucky's Software", Url:="‹www.ChuckySoft.com")› \_

Public Class VbNetSnapIn Implements IAppFunctionality

 Public Sub DoIt() Implements CommonSnappableTypes.IAppFunctionality.DoIt

  MessageBox.Show("Вы только что подключили блок VB .NET!")

 End Sub

End Class

Говорить здесь особенно не о чем. Однако обратите внимание на то, что синтаксис применения атрибутов в Visual Basic .NET предполагает использование *угловых* (‹›), а не квадратных ([]) скобок.

**Создание расширяемого приложения Windows Forms**

Заключительным шагом будет создание приложения Windows Forms, которое позволит пользователю выбрать подключаемый блок с помощью стандартного диалогового окна открытия файла Windows. Создав новое приложение Windows Forms (с именем MyExtendableApp), добавите ссылку на компоновочный блок CommonSnappableTypes.dll, но*не устанавливайте*ссылок на библиотеки программного кода CSharpSnapIn.dll и VbNetSnapIn.dll. Помните о том, что целью создания этого приложения является демонстрация динамического связывания и отображения при выяснении возможности подключения независимых двоичных блоков, созданных сторонними производителями.

Снова подчеркнем, что здесь не рассматриваются детали процесса построения приложений Windows Forms. Тем не менее, предполагается, что вы поместите компонент MenuStrip в окно формы и определите с его помощью меню Сервис, которое будет содержать единственный пункт Подключаемый модуль (рис. 12.11).

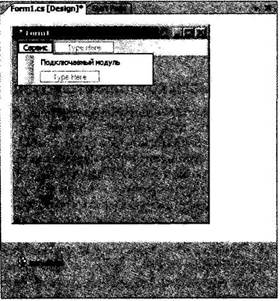


Рис. 12.11. Исходный графический интерфейс MyExtendableApp

Эта форма Windows должна также содержать тип Listbox (которому здесь назначено имя lstLoadedSnapIns), используемый для отображения имен подключаемых модулей, загружаемых пользователем. На рис. 12.12 показан окончательный вид графического интерфейса приложения, о котором идет речь.

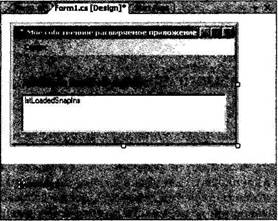


Рис. 12.12. Окончательный вид графического интерфейса MyExtendableApp

Программный код для обработки выбора Сервис→Подключаемый модуль из меню (этот программный код можно создать с помощью двойного щелчка на пункте меню в окне проектирования формы), отображает диалоговое окно Открытие файла и читает путь к выбранному файлу. Соответствующая строка пути затем посылается вспомогательной функции LoadExternalModule() для обработки. Метод возвращает false (ложь), если он не обнаруживает класс, реализующий IAppFunctionality.

private void snapInModuleToolStripMenuItem\_Click(object sender, EventArgs e) {

**// Позволяет пользователю выбрать компоновочный блок для загрузки.**

 OpenFileDialog dlg = new OpenFileDialog();

 if (dlg.ShowDialog() == DialogResult.OK) {

  if (LoadExternalModule(dlg.FileName) == false) MessageBox.Show("Нет реализации IAppFunctionality!");

 }

}

Метод LoadExternalModule() решает следующие задачи.

• Динамически загружает компоновочный блок в память.

• Выясняет, содержит ли компоновочный блок тип, реализующий IAppFunctionality

Если тип, реализующий IAppFunctionality, обнаружен, вызывается метод DoIt(), и абсолютное имя типа добавляется в Listbox (заметьте, что цикл for должен выполнить проход по всем типам компоновочного блока, чтобы учесть возможность наличия в одном компоновочном блоке нескольких модулей расширения).

private bool LoadExternalModule(string path) {

 bool foundSnapIn = false;

 IAppFunctionality itfAppFx;

**// Динамическая загрузка выбранного компоновочного блока.**

 Assembly theSnapInAsm = Assembly.LоadFrom(path);

**// Получение всех типов компоновочного блока.**

 Tуре[] theTypes = theSnapInAsm.GetTypes();

**// Поиск типа с реализацией IAppFunctionality.**

 for (int i = 0; i ‹ theTypes.Length; i++) {

  Type t = theTypes[i].GetInterface("IAppFunctionality");

  if (t != null) {

   foundSnapIn = true;

**// Динамическое связывание для создания типа.**

   object о = theSnapInAsm.CreateInstance(theTypes[i].FullName);

**// Вызов DoIt() через интерфейс.**

   itfAppFx = о as IAppFunctionality;

   itfAppFx.DoIt();

   lstLoadedSnapIns.Items.Add(theTypes[i].FullName);

  }

 }

 return foundSnapIn;

}

Теперь вы можете выполнить свое приложение. При выборе компоновочных блоков CSharpSnapIn.dll и VbNetSnapIn.dll вы должны увидеть соответствующее сообщение. На рис. 12.13 показан один из возможных вариантов выполнения.



Рис. 12.13. Подключение внешних компоновочных блоков

Завершающей задачей будет отображение метаданных, соответствующих атрибуту [CompanyInfo]. Для этого просто обновите LoadExternalModule(), чтобы перед выходом из контекста if вызывалась новая вспомогательная функция DisplayCompanyData(). Эта функция имеет один параметр типа System.Type.

private bool LoadExternalModule(string path) {

 …

 if (t != null) {

  …

**// Отображение информации о компании.**

  DisplayCompanyData(theTypes[i]);

 }

 return foundSnapIn;

}

Для поступающего на вход типа просто отобразите атрибут [CompanyInfo].

private void DisplayCompanyData(Type t) {

**// Получение данных [CompanyInfo].**

 object[] customAtts = t.GetCustomAttributes(false);

**// Вывод данных.**

 foreach (CompanyInfoAttribute с in customAtts) {

  MessageBox.Show(с.Url, String.Format("Дополнительные сведения о {0} ищите по адресу", с.Name));

 }

}

Превосходно! На этом рассмотрение примера завершается. Я надеюсь, что к этому моменту нашего обсуждения вы осознаете, что подходы, представленные в этой главе, могут найти применение и на практике, а не только при разработке средств построения программ.

**Исходный код.**Программный код приложений CommonSnappableTypes, CSharpSnapIn, VbNetSnapIn и MyExtendableApp размещен в подкаталоге, соответствующем главе 12.

**Резюме**

Сервис отображения оказывается весьма интересным аспектом построения надежного окружения при использовании объектно-ориентированного подхода. В среде .NET ключевыми элементами сервиса отображения являются тип System.Туре и пространство имен System.Reflection. Отображение представляет собой процесс выяснения основных характеристик и возможностей типа в среде выполнения.

Динамическое связывание предполагает создание типа и вызов его членов при отсутствий априорной информации о конкретных именах соответствующих членов. Как показывает пример создаваемого в этой главе расширяемого приложения, это очень мощная техника, которая может использоваться как создателями инструментов разработки приложений, так и пользователями этик инструментов, В главе была также рассмотрена роль программирования с помощью атрибутов. При добавлении атрибутов в определения типов добавляется соответствующая информация в метаданные компоновочного блока.

**ГЛАВА 13. Процессы, домены приложений, контексты и хосты CLR**

В предыдущих двух главах мы рассмотрели шаги, которые предпринимаются в среде CLR при выяснении параметров размещения внешних компоновочных блоков, а также роль метаданных .NET. В этой главе мы рассмотрим подробности того, как CLR обрабатывает компоновочный блок, и попытаемся понять взаимосвязь между процессами, доменами приложений и контекстами объектов.

В сущности, *домены приложений*(или *АррDотаin*) представляют собой логические подразделы в рамках данного процесса, содержащие наборы связанных компоновочных блоков .NET. Вы увидите, что домены приложений разделяются контекстными границами, которые используются для группировки подобных .NET-объектов. С использованием понятия контекста среда CLR подучает возможность гарантировать, что объекты с особыми требованиями к среде выполнения будут обрабатываться надлежащим образом.

Обладая пониманием того, как среда CLR обрабатывает компоновочный блок, мы с вами сможем выяснить, что такое хостинг CLR. Как уже говорилось в главе 1, сама среда CLR представляется (по крайней мере, отчасти) файлом mscoree.dll. При запуске выполняемого компоновочного блока файл mscoree.dll загружается автоматически, но, как вы сможете убедиться, в фоновом режиме при этом выполняется целый ряд шагов, скрытых от глаз пользователя.

**Выполнение традиционных процессов Win32**

Понятие "процесс" существовало в операционных системах Windows задолго до появления платформы .NET. Упрощенно говоря, термин *процесс*используется для обозначения множества ресурсов (таких, как внешние библиотеки программного кода и первичный поток) и выделяемой памяти, необходимых для работы приложения. Для каждого загруженного в память файла \*.ехe операционная система создает отдельный и изолированный процесс, используемый в течение всего "жизненного цикла" соответствующего приложения. В результате такой изоляции приложений повышается надежность и устойчивость среды выполнения, поскольку отказ в ней одного процесса не влияет на функционирование другого.

Каждому процессу Win32 назначается уникальный идентификатор PID (Process ID – идентификатор процесса), и процесс, при необходимости, может независимо загружаться или выгружаться операционной системой (или программными средствами с помощью вызовов Win32 API). Вы, возможно, знаете, что на вкладке Процессы в окне Диспетчер задан Windows (которое можно активизировать нажатием комбинации клавиш ‹Ctrl+Shift+Esc>) можно увидеть информацию о процессах, выполняющихся на машине, включая информацию PID и имя образа (рис. 13.1).

**Замечание.**Если столбец PID в окне Диспетчер задач Windows не отображается, выберите в этом окне команду Вид→Выбрать столбцы… из меню и в открывшемся после этого окне установите флажок Идентификатор процесса (PID).

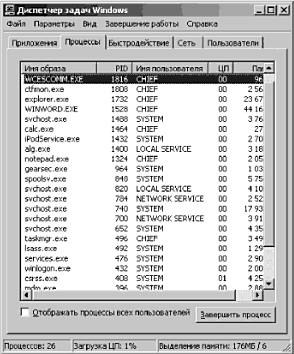


Рис. 13.1. Диспетчер задач Windows

**Обзор потоков**

Каждый процесс Win32 имеет один главный "поток", выполняющий функции точки входа в приложение. В следующей главе будет выяснено, как создавать дополнительные потоки и соответствующий программный код, применяя возможности пространства имен System.Threading, но пока что для освещения вопросов, представленных здесь, нам нужно выполнить определенную вспомогательную работу. Во-первых, заметим, что поток – это "нить" выполнения в рамках данного процесса. Первый поток, созданный точкой входа процесса, называется *первичным потоком.*Приложения Win32 с графическим интерфейсом пользователя определяют в качестве точки входа приложения метод WinMain(). Консольные приложения для этой цели используют метод Main(). Процессы, состоящие из одного первичного потока, будут *потокоустойчивыми,*поскольку в них в каждый момент времени только один поток может получить доступ к данным приложения. Однако одно-поточный процесс (особенно если он основан на графическом интерфейсе) часто бывает "склонен" не отвечать пользователю при выполнении потоком достаточно сложных действий (например, связанных с печатью большого текстового файла, запутанными вычислениями или попытками соединиться с удаленным сервером).

Учитывая эти потенциальные недостатки однопоточных приложений, Win32 API позволяет первичным потокам порождать дополнительные вторичные потоки (также называемые *рабочими потоками*), используя, например, такую удобную функцию Win32 API, как CreateThread(). Каждый поток (первичный или вторичный) в таком процессе становится уникальным элементом выполнения и получает доступ ко всем открытым элементам данных на условиях конкуренции.

Вы можете сами догадаться, что разработчики создают дополнительные потоки, как правило, для того, чтобы программа могла быстрее реагировать на. действия пользователя. Многопоточные процессы обеспечивают иллюзию того, что все действия выполняются приблизительно за одно и то же время. Например, приложение может порождать рабочий поток для выполнения действий, требующих много времени (снова вспомним о печати большого файла). Как только созданный вторичный поток начинает свою работу, главный поток снова получает возможность отвечать на пользовательский ввод, что позволяет процессу в целом потенциально обеспечить лучшую производительность. Хотя в реальности этого может и не произойти: использование слишком большого числа потоков в рамках одного процесса может *ухудшить*производительность, поскольку процессору придется часто переключаться между активными потоками (а это требует немало времени).

В реальности всегда следует учитывать то, что многопоточность является, по сути, иллюзией, обеспечиваемой операционной системой. Машины, основанные на одним процессоре, в действительности не способны обрабатывать несколько потоков одновременно. Вместо этого системы с одним процессором выделяют каждому потоку свою) часть времени (что называют *квантованием*времени) на основе уровня приоритета данного потока. Когда квант времени потока заканчивается, текущий поток приостанавливается, чтобы свою задачу мог выполнить другой поток. Чтобы поток "помнил", что происходило перед тем, как поток был временно "отодвинут в сторону", каждый поток получает возможность записать необходимые данные в блик TLS (Thread Local Storage – локальная память потока), и каждому потоку обеспечивается отдельный стек вызовов, как показано на рис. 13.2.

Если тема потоков для вас нова, не слишком беспокойтесь о деталях. На этот момент достаточно запомнить только то, что поток является уникальной "нитью" выполнения в рамках процесса Win32. Каждый процесс имеет первичный поток (создаваемый точкой входа в приложение) и может содержать дополнительные потоки, которые создаются программными средствами.

**Замечание**. Новые процессоры Intel имеют особенность, называемую технологией НТ (Hyper-Threading Technology – гиперпотоковая технология), которая позволяет одному процессору при определенных условиях обрабатывать множество потоков одновременно. Подробности описания этой технологии можно найти по адресу <http://www.intel.com/info/hyperthreading>.

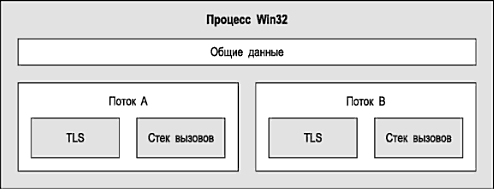


Рис. 13.2. Взаимосвязь процесса и потоков Win32

**Взаимодействие с процессами в рамках платформы .NET**

Хотя процессы и потоки сами по себе не являются чем-то новым, способы взаимодействия с этими примитивами в рамках платформы .NET существенно изменены (к лучшему). Чтобы успешно пройти путь к пониманию приемов построения компоновочных блоков с поддержкой множества потоков (см. главу 14), мы начнем с обсуждения возможностей взаимодействия с процессами на основе использования библиотек базовых классов .NET.

Пространство имен System.Diagnostics определяет ряд типов, позволяющих программное взаимодействие с процессами, а также типов, связанных с диагностикой системы (например, с журналом регистрации системных событий и счетчиками производительности). В этой главе мы рассмотрим только те связанные с процессами типы, которые определены в табл. 13.1.

**Таблица 13.1.**Избранные члены пространства имен System.Diagnostics

|  |  |
| --- | --- |
| **Типы System.Diagnostics для поддержки процессов** | **Описание** |
| Process | Класс Process обеспечивает доступ к локальным и удаленным процессам, а также позволяет программно запускать и останавливать процессы |
| ProcessModule | Этот тип представляет модуль (\*.dll или \*.exe), загруженный в рамках конкретного процесса. При этом тип ProcessModule может представлять *любой*модуль – модуль COM, модуль .NET или традиционный двоичный файл C |
| ProcessModuleCollection | Предлагает строго типизованную коллекцию объектов ProcessModule |
| ProcessStartlnfo | Указывает множество значений, используемых при запуске процесса с помощью метода Process.Start() |
| ProcessThread | Представляет поток в рамках данного процесса. Тип ProcessThread используется для диагностики множества потоков процесса, а не для того, чтобы порождать новые потоки выполнения в рамках данного процесса |
| ProcessThreadCollection | Предлагает строго типизованную коллекцию объектов PrосessThread |

Тип System.Diagnostics.Process позволяет проанализировать процессы, выполняемые на данной машине (локальной или удаленной). Класс Process предлагает также члены, которые позволяют запускать и останавливать процессы программными средствами, устанавливать уровни приоритета и получать список активных потоков и/или загруженных модулей, выполняемых в рамках данного процесса. В табл. 13.2 предлагается список некоторых (но не всех) членов System.Diagnostics.Process.

**Таблица 13.2.** Избранные члены типа Process

|  |  |
| --- | --- |
| **Член** | **Описание** |
| ExitCode | Свойство, содержащее значение, которое указывается процессом при завершении его работы. Для получения этого значения необходимо обработать событие Exited (при асинхронном уведомлении) или вызвать метод WaitForExit() (при синхронном уведомлении) |
| ExitTime | Свойство, содержащее штамп времени, соответствующий прекращению работы процесса (и представленный типом DateTime) |
| Handle | Свойство, возвращающее дескриптор, назначенный процессу операционной системой |
| HandleCount | Свойство, возвращающее число дескрипторов, открытых процессом |
| Id | Свойство, содержащее идентификатор процесса (PID) для данного процесса |
| MachineName | Свойство, содержащее имя компьютера, на котором выполняется данный процесс |
| MainModule | Свойство, получающее тип ProcessModule, который представляет главный модуль данного процесса |
| MainWindowTitle MainWindowHandle | Свойство MainWindowTitle получает заголовок главного окна процесса (если процесс не имеет главного окна, будет возвращена пустая строка). Свойство MainWindowHandle получает дескриптор (представленный типом System.IntPtr) соответствующего окна. Если процесс не имеет главного окна, типу IntPtr присваивается значение System.IntPtr.Zero |
| Modules | Свойство, обеспечивающее доступ к строго типизованной коллекции ProcessModuleCollection, представляющей множество модулей (\*.dll или \*.exe), загруженных в рамках текущего процесса |
| PriorityBoostEnabled | Это свойство указывает, должна ли операционная система временно ускорять выполнение процесса, когда его главное окно получает фокус ввода |
| PriorityClass | Свойство, позволяющее прочитать или изменить данные базового приоритета соответствующего процесса |
| ProcessName | Свойство, содержащее имя процесса (которое, как вы можете догадаться, соответствует имени приложения) |
| Responding | Значение этого свойства указывает, должен ли пользовательский интерфейс процесса реагировать на действия пользователя |
| StartTime | Свойство с информацией о времени, соответствующем старту данного процесса (эта информация представлена типом DateTime) |
| Threads | Свойство, получающее набор потоков, выполняющихся в рамках данного процесса (представляется массивом типов ProcessThread) |
| CloseMainWindow() | Метод, завершающий процесс с пользовательским интерфейсом путем отправки соответствующего сообщения о закрытии главного окна |
| GetCurrentProcess() | Статический метод, возвращающий тип Process, используемый для представления процесса, активного в настоящий момент |
| GetProcesses() | Статический метод, возвращающий массив компонентов Process, выполняющихся на данной машине |
| Kill() | Метод, немедленно прекращающий выполнение соответствующего процесса |
| Start() | Метод, начинающий выполнение процесса |

**Список выполняемых процессов**

Чтобы привести пример обработки типов Process, предположим, что у нас есть консольное приложение C# ProcessManipulator, которое определяет следующий вспомогательный статический метод.

public static void ListAllRunningProcesses() {

**// Получение списка процессов, выполняемых на данной машине.**

 Process[] runningProcs = Process.GetProcesses(".");

**// Печать значения PID и имени каждого процесса.**

 foreach(Process p in runningProcs) {

  string info = string.Format("-› PID: {0}\tИмя: {1}", p.Id, p.ProcessName);

  Console.WriteLine(info);

 }

 Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

Обратите внимание на то, что статический метод Process.GetProcesses() возвращает массив типов Process, представляющих процессы, запущенные на выполнение на целевой машине (используемая здесь точка обозначает локальный компьютер).

После получения массива типов Process можно использовать любой из членов, приведенных в табл. 13.2. Здесь просто отображается значение PID и имя каждого из процессов. В предположении о том, что вы обновили метод Main() для вызова ListAllRunningProcesses(), в результате выполнения соответствующей программы вы должны увидеть нечто подобное показанному на рис. 13.3.

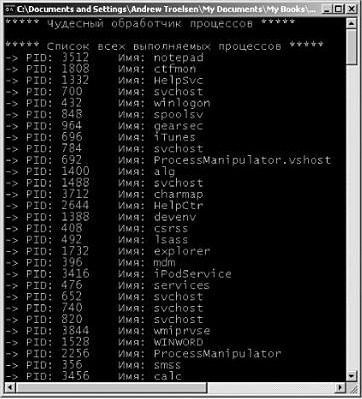


Рис. 13.3. Перечень запущенных процессов

**Чтение данных конкретного процесса**

В дополнение к полному списку всех запущенных на данной машине процессов, статический метод Process.GetProcessById() позволяет прочитать данные отдельного процесса по его значению PID. Если запросить доступ к процессу по несуществующему значению PID, будет сгенерировано исключение ArgumentException. Так, чтобы получить объект Process, представленный значением PID, равным 987, можно написать следующее.

**// Если процесса с PID=987 нет, то среда выполнения**

**// сгенерирует соответствующее исключение.**

static void Main(string[] args) {

 Process theProc;

 try {

  theProc = Process.**GetProcessByld**(987);

 } catch { **// Общий блок catch для простоты.**

  Console.WriteLine("-› Извините, некорректное значение PID!");

 }

}

**Список множества потоков процесса**

Тип класса Process обеспечивает и способ программного получения множества всех потоков, используемых данным потоком в настоящий момент. Множество потоков представляется строго типизованной коллекцией ProcessThreadCollection, которая содержит соответствующий набор отдельных типов ProcessThread. Для примера предположим, что в наше текущее приложение была добавлена следующая вспомогательная статическая функция.

public static void EnumThreadsForPid(int pID) {

 Process theProc;

 try {

  theProc = Process.GetProcessById(pID);

 } catch {

  Console.WriteLine("-› Извините, некорректное значение PID!");

  Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

  return;

 }

**// Вывод информации для каждого потока указанного процесса.**

 Console.WriteLine("Это потоки, выполняемые в рамках {0}", theProc.ProcessName);

 ProcessThreadCollection theThreads = theProc.**Threads;**

foreach (ProcessThread pt in theThreads) {

 string info = string.Format("-› ID: {0}\tBpeмя запуска {1}\tПриоритет {2}", pt.Id, pt.StartTime.ToShortTimeString(), pt.PriorityLevel);

 Console.WriteLine(info);

}

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n").

}

Как видите, свойство Threads типа System.Diagnostics.Process обеспечивает доступ к классу ProcessThreadCollection. Здесь для каждого потока в рамках указанного клиентом процесса выводится назначенный потоку идентификатор ID, время запуска и приоритет. Обновите метод Main() программы для запроса у пользователя значения PID процесса так, как показано ниже.

static void Main(string[] args) {

 …

**// Запрос PID у пользователя и вывод списка активных потоков.**

 Console.WriteLine("\*\*\*\*\* Введите значение PID процесса \*\*\*\*\*");

 Console.Write("PID: ");

 string pID = Console.ReadLine();

 int theProcID = int.Parse(pID);

 EnumThreadsForPid(theProcID);

 Console.ReadLine();

}

В результате выполнения обновленной программы вы должны получить вывод, подобный показанному на рис. 13.4.

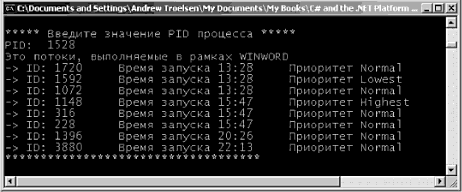


Рис. 13.4*.*Перечень потоков в рамках выполняемого процесса

Кроме членов Id, StartTime и PriorityLevel, тип ProcessThread имеет и другие члены, которые могут представлять интерес. Некоторые из таких членов приведены в табл. 13.3.

Таблица 13.3. Подборка членов типа ProcessThread

|  |  |
| --- | --- |
| **Член** | **Описание** |
| BasePriority | Читает значение базового приоритета потока |
| CurrentPriority | Читает значение текущего приоритета потока |
| Id | Читает уникальный идентификатор потока |
| IdealProcessor | Задает предпочтительный процессор для выполнения данного потока |
| PriorityLevel | Читает или задает уровень приоритета для данного потока |
| ProcessorAffinity | Задает процессоры, на которых может выполняться ассоциированный поток |
| StartAddress | Читает адрес в памяти для функции, которая вызывалась операционной системой для запуска данного потока |
| StartTime | Читает информацию о времени запуска данного потока операционной системой |
| ThreadState | Читает информацию о текущем состоянии потока |
| TotalProcessorTime | Читает общую оценку времени, в течение которого данный поток использовал процессор |
| WaitReason | Читает информацию о причине, по которой поток находится в ожидании |

Перед тем как двигаться дальше, следует заметить, что тип ProcessThread не *является*тем элементом, который можно использовать для создания, остановки или ликвидации потоков в рамках платформы .NET. Тип ProcessThread является средством получения диагностической информации об активных потоках Win32 в рамках выполняющихся процессов. То, как строить многопоточные приложения с помощью пространства имен System.Threading, мы с вами выясним в главе 14.

**Информация о наборе модулей процесса**

Теперь выясним, как выполнить цикл по всем модулям, загруженным в рамках данного процесса. Напомним, что *модуль*- это общее название, используемое для обозначения \*.dll (или \*.exe). При доступе к ProcessModuleCollection с помощью свойства Process.Module вы получаете перечень *всех модулей,*задействованных в рамках соответствующего процесса – модулей .NET, модулей COM и традиционных библиотек C. Рассмотрите следующую вспомогательную функцию, которая перечислит модули конкретного процесса, заданного с помощью PID.

public static void EnumModsForPid(int pID) {

 Process theProc;

 try {

  theProc = Process.GetProcessById(pID);

 } catch {

  Console.WriteLine("-› Извините, некорректное значение PID!");

  Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

  return;

 }

 Console.WriteLine("Загруженные модули для {0}:", theProc.ProcessName);

 try {

  ProcessModuleCollection theMods = theProc.**Modules;**

  foreach (ProcessModule pm in theMods) {

   string info = string.Format("-› Имя модуля: {0}", pm.ModuleName);

   Console.WriteLine(info);

  }

  Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

 } catch {

  Console.WriteLine("Модулей не обнаружено!");

 }

}

Чтобы увидеть пример возможного вывода программы, давайте проверим затрушенные модули для. процесса, выполняемого в рамках рассматриваемого здесь консольного приложения ProcessManipulator. Для этого запустите приложение, выясните значениеPID, соответствующее ProcessManipulator.exe, и передайте это значение методу EnumModsForPid() (не забудьте соответствующим образом обновить метод Main(). Вы, наверное, удивитесь, увидев весь список модулей \*.dll, которые используются для такого простого консольного приложения (atl.dll, mfc42u.dll, oleaut32.dll и т.д.). На рис. 13.5 показан результат запуска.

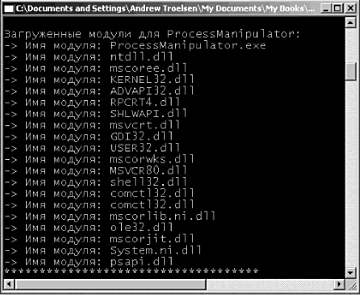


Рис. 13.5. Перечень загруженных модулей в рамках выполняющегося процесса

**Начало и остановка процессов с помощью программных средств**

В завершение этого раздела мы рассмотрим методы Start() и Kill() типа System.Diagnostics.Process. По именам этих методов вы можете догадаться, что они обеспечивают, соответственно, программный запуск и программное завершение процесса. Рассмотрите, например, вспомогательный статический метод StartAndKillProcess().

public static void StartAndKillProcess() {

**// Запуск Internet Explorer.**

 Process ieProc = Process.Start("IExplore.exe", "www.intertechtraining.com");

 Console.Write("-› Нажмите ‹Enter›, чтобы завершить {0}…", ieProc.ProcessName);

 Console.ReadLine();

**// Завершение процесса iexplorer.exe.**

 try {

  ieProc.Kill();

 } catch {}**// Если пользователь уже завершил процесс.…**

}

Статический метод Process.Start() является перегруженным. Как минимум, вы должны указать имя процесса, который следует запустить (например, Microsoft Internet Explorer). В этом примере используется вариация метода Start(), позволяющего указать любые дополнительные аргументы, передаваемые точке входа программы (т.е. методу Main()).

Метод Start(), кроме того, позволяет передать тип System.Diagnostics. ProcessStartInfo, чтобы указать дополнительную информацию о том, как должен стартовать данный процесс. Вот формальное определение ProcessStartInfo (подробности можно найти в документации .NET Framework 2.0 SDK).

public sealed class System.Diagnostics. ProcessStartInfo : object {

 public ProcessStartInfo();

 public ProcessStartInfo(string fileName);

 public ProcessStartInfo(string fileName, string arguments);

 public string Arguments { get; set; }

 public bool CreateNoWindow { get; set; }

 public StringDictionary EnvironmentVariables { get; }

 public bool ErrorDialog { get; set; }

 public IntPtr ErrorDialogParentHandle { get; set; }

 public string FileName { get; set; }

 public bool RedirectStandardError { get; set; }

 public bool RedirectStandardInput { get; set; }

 public bool RedirectStandardOutput { get; set; }

 public bool UseShellExecute { get; set; }

 public string Verb { get; set; }

 public string[] Verbs { get; }

 public ProcessWindowStyle WindowStyle { get; set; }

 public string WorkingDirectory { get; set; }

 public virtual bool Equals(object obj);

 public virtual int GetHashCode();

 public Type GetType();

 public virtual string ToString();

}

Независимо от того, какую версию метода Process.Start() вы вызовете, будет возвращена ссылка на новый активизированный процесс. Чтобы завершить выполнение процесса, просто вызовите метод Kill() уровня экземпляра.

**Исходный код**. Проект ProcessManipulator размещен в подкаталоге, соответствующем главе 13.

**Домены приложений .NET**

Теперь, когда вы понимаете роль процессов Win32 и возможностей взаимодействия с ними средствами управляемого программного кода, давайте рассмотрим понятие домена приложения .NET. По правилам платформы .NET компоновочные блоки не размещаются в рамках процесса непосредственно (как это было в традиционных приложениях Win32). Вместо этого выполняемый файл .NET помещается в обособленный логической раздел процесса, называемый *доменом приложения*(сокращенно AppDomain). Вы увидите, что один процесс может содержать множество доменов приложения, каждый из которых будет обслуживать свой выполняемый файл .NET. Такое дополнительное разделение традиционного процесса Win32 дает определенные преимущества, и некоторые из них указаны ниже.

• Домены приложения являются ключевым аспектом независимой от ОС природы платформы .NET, поскольку такое логическое деление абстрагируется от того, как именно ОС представляет загруженный выполняемый объект.

• Домены приложения являются существенно менее ресурсоемкими в отношении времени процессора и памяти, чем весь процесс в целом. Поэтому среда CLR способна загружать и выгружать домены приложений намного быстрее, чем формальный процесс.

• Домены приложения обеспечивают лучший уровень изоляции для загруженного приложения. Если один домен приложения в рамках процесса "терпит неудачу", остальные домены приложений могут продолжать функционировать.

Из приведенного списка следует, что один процесс может содержать любое число доменов приложения, каждый из которых полностью изолирован от других доменов приложения в рамках данного процесса (а также любого другого процесса). С учетом этого следует понимать, что приложение, выполняющееся в одном домене приложения, не может получить данные (в частности, значения глобальных переменных или статических полей) другого домена приложений иначе, как с помощью протокола удаленного взаимодействия .NET (который мы рассмотрим в главе 18).

Хотя один процесс и *может*принять множество доменов приложения, так бывает не всегда. Как минимум, процесс ОС будет содержать то, что обычно называют *доменом приложения, созданным по умолчанию.*Этот специальный домен приложения автоматически воздается средой CLR во время запуска процесса.

После этого CLR создает дополнительные домены приложения по мере необходимости, Если потребуется (хотя это и маловероятно), вы можете программно создавать домены приложения в среде выполнения в рамках выполняемого процесса, используя статические методы класса System.AppDomain. Этот класс оказывается также полезным для осуществления низкоуровневого контроля доменов приложения. Основные члены этого класса описаны в табл. 13.4.

Кроме того, тип AppDomain определяет небольшой набор событий, соответствующих различным моментам цикла существования домена приложения (табл. 13.5).

**Таблица 13.4.**Основные члены класса AppDomain

|  |  |
| --- | --- |
| **Член** | **Описание** |
| CreateDomain() | Статический метод, с помощью которого создается новый домен приложения в данном процессе. Среда CLR сама создает новые домены приложения по мере необходимости, поэтому вероятность того, что вам понадобится вызывать этот член, близка к нулю |
| GetCurrentThreadId() | Статический метод, возвращающий ID активного потока в данном домене приложения |
| Unload() | Еще один статический метод, позволяющий выгрузить указанный домен приложения для данного процесса |
| BaseDirectory | Свойство, возвращающее базовый каталог, используемый при поиске зависимых компоновочных блоков |
| CreateInstance() | Метод, создающий экземпляр указанного типа, определенного в указанном файле компоновочного блока |
| ExecuteAssembly() | Метод, выполняющий компоновочный блок в рамках домена приложения, заданного именем файла |
| GetAssemblies() | Метод, который читает список компоновочных блоков .NET, загруженных в данном домене приложения (двоичные файлы COM и C игнорируются) |
| Load() | Метод, используемый для динамической загрузки компоновочного блока в рамках данного домена приложения |

**Таблица 13.5.**События типа AppDomain

|  |  |
| --- | --- |
| **Событие** | **Описание** |
| AssemblyLoad | Возникает при загрузке компоновочного блока |
| AssemblyResolve | Возникает, когда не удается идентифицировать компоновочный блок |
| DomainUnload | Возникает перед началом выгрузки домена приложения |
| ProcessExit | Возникает для домена приложения, созданного по умолчанию, когда завершается родительский процесс этого домена |
| ResourceResolve | Возникает, когда не удается идентифицировать ресурс |
| TypeResolve | Возникает, когда не удается идентифицировать тип |
| UnhandledException | Возникает, когда остается без обработки сгенерированное исключение |

**Список доменов приложения процесса**

Для примера программного взаимодействия с доменами приложений .NET предположим, что у нас есть новое консольное приложение C# с именем AppDomainManipulator, в рамках которого определяется статический метод PrintAllAssembliesInAppDomain(). Этот вспомогательный метод использует AppDomain.GetAssemblies(), чтобы получить список всех двоичных файлов .NET, выполняющихся в рамках данного домена приложения.

Соответствующий список представляется массивом типов System.Reflection. Assembly, поэтому необходимо использовать пространство имен System. Reflection (см. главу 12). Получив массив компоновочных блоков, вы выполняете цикл по элементам массива и печатаете понятное имя и версию каждого модуля.

public static void PrintAllAssembliesInAppDomain(ApsDomain ad) {

 Assembly[] loadedAssemblies = ad.GetAssemblies();

 Console.WriteLine("\*\*\* Компоновочные блоки в рамках {0} \*\*\*\n", ad.FriendlyName);

 foreach (Assembly a in loadedAssemblies) {

  Console.WriteLine("-*›*Имя: {0}", a.GetName().Name);

  Console.WriteLine("-› Версия: {0}\n", a.GetName().Version);

 }

}

Теперь обновим метод Main(), чтобы перед вызовом PrintAllAssembliesInAppDomain() получить ссылку на текущий домен приложения, используя свойство AppDomain.CurrentDomain.

Чтобы сделать пример более интересным, метод Main() открывает окно сообщения Windows Forms (для этого среда CLR должна загрузить компоновочные блоки System.Windows.Forms.dll, System.Drawing.dll и System.dll, так что не забудьте установить ссылки на эти компоновочные блоки и соответственно изменить набор операторов using).

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Чудесное приложение AppDomain \*\*\*\*\*\n");

**// Чтение информации для текущего AppDomain.**

 AppDomain defaultAD= AppDomain.CurrentDomain;

 MessageBox.Show(''Привет");

 PrintAllAssembliesInAppDomain(defaultAD);

 Console.ReadLine();

}

На рис. 13.6 показан соответствующий вывод (номера версий у вас могут быть другими).

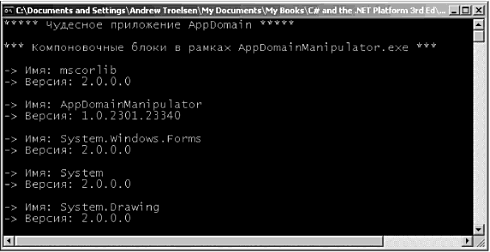


Рис. 13.6. Перечень компоновочных блоков в рамках текущего домена приложений

**Программное создание новых доменов приложения**

Напомним, что один процесс может содержать множество доменов приложения, И хотя вам в программном коде вряд ли понадобится вручную создавать домены приложения, вы имеете возможность сделать это с помощью статического метода CreateDomain(). Нетрудно догадаться, что метод AppDomain.CreateDomain() перегружен. Вы, по крайней мере, должны указать понятное имя нового домена приложения, как показано ниже.

static void Main(string[] args) {

 …

**// Создание нового AppDomain в рамках текущего процесса.**

 AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");

 PrintAllAssembliesInAppDomain(anotherAD);

 Console.ReadLine();

}

Если выполнить приложение теперь (рис. 13.7), вы увидите, что компоновочные блоки System.Windows.Forms.dll, System.Drawing.dll и System.dll будут загружены только в рамках домена приложения, созданного по умолчанию. Это может показаться нелогичным для тех. кто имеете опыт программирования с использованием традиционных подходов Win32 (скорее, оба домена приложения должны иметь доступ к одному и тому же множеству компоновочных блоков). Напомним, однако, о том. что компоновочный блок загружается в рамки *домена приложения*а не в рамки непосредственно самого процесса.

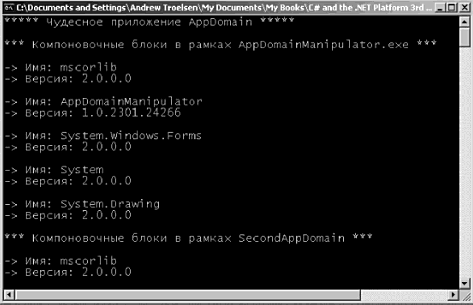


Рис. 13.7. Один процесс с двумя доменами приложения

Далее, обратите внимание на то, что домен приложения SecondAppDomain автоматически получает свою собственную копию mscorlib.dll, поскольку этот ключевой компоновочный блок автоматически загружается средой CLR для каждого домена приложения. Это порождает следующий вопрос "Как можно программно загрузить компоновочный блок в домен приложения?" Ответ: с помощью метода АррDomain.Load() (или, альтернативно, с помощью AppDomain.executeAssembly()). В предположении, что вы скопировали CarLibrary.dll в каталог приложения AppDomainManipulator.exe, вы можете загрузить CarLibrary.dll в домен приложения SecondAppDomain так.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Чудесное приложение AppDomain \*\*\*\*\*\n");

 …

**// Загрузка CarLibrary.dll в новый AppDomain.**

 AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");

 anotherAD.Load("CarLibrary");

 PrintAllAssembliesInAppDomain(anotherAD);

 Console.ReadLine();

}

Чтобы закрепить понимание взаимосвязей между процессами, доменами приложения и компоновочными блоками, рассмотрите рис. 13.8, на котором представлена диаграмма внутреннего устройства только что построенного процесса AppDomainManipulator.exe.

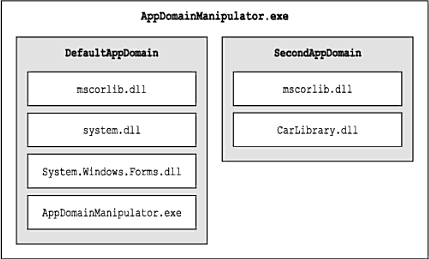


Рис. 13.8. Схема функционирования процесса AppDomainManipulator.exe

**Программная выгрузка доменов приложения**

Важно понимать, что среда CLR не позволяет выгружать отдельные компоновочные блоки .NET. Однако, используя метод AppDomain.Unload(), вы можете избирательно выгрузить домен приложения из объемлющего процесса. При этом домен приложения выгрузит по очереди каждый компоновочный блок.

Напомним, что тип AppDomain определяет набор событий, одним из которых является DomainUnload. Это событие генерируется тогда, когда домен приложения (не являющийся доменом, созданным по умолчанию) выгружается из содержащего этот домен процесса. Другим заслуживающим внимания событием является событие ProcessExit, которое генерируется при выгрузке из процесса домена, создаваемого по умолчанию (что, очевидно, влечет за собой завершение всего процесса). Так, если вы хотите программно выгрузить anotherAD из процесса AppDomainManipulator.exe и получить извещение о том, что соответствующий домен приложения закрыт, можете использовать следующую программную логику событий.

static void Main(string[] args) {

 …

**// Привязка к событию DomainUnload**.

 anotherAD.**DomainUnload**+= new EventHandler(anotherAD\_DomainUnload);

**// Теперь выгрузка anotherAD.**

AppDomain.**Unload**(anotherAD);

}

Обратите внимание на то, что событие DomainUnload работает в паре с делегатом System.EventHandler, поэтому формат anotherAD\_DomainUnload() требует следующих аргументов.

public static void anotherAD\_DomainUnload(object sender, EventArgs e) {

 Console.WriteLine("\*\*\*\*\* Выгрузка anotherAD! \*\*\*\*\*\n");

}

Если вы хотите получить извещение при выгрузке домена приложения, созданного по умолчанию, измените метод Main() так, чтобы обработать событие ProcessEvent, соответствующее домену приложения по умолчанию:

static void Main(string [] args) {

 …

 AppDomain defaultAD = AppDomain.CurrentDomain;

 defaultAD.**ProcessExit**+=new EventHandler(defaultAD\_ProcessExit);

}

и определите подходящий обработчик событий.

private static void defaultAD\_ProcessExit (object sender, EventArgs e) {

 Console.WriteLine("\*\*\*\*\* Выгрузка defaultAD! \*\*\*\*\*\n");

}

**Исходный код.**Проект AppDomainManipulator размещен в подкаталоге, соответствующем главе 13.

**Границы контекста объекта**

Итак, вы могли убедиться, что домены приложения – это логические разделы в рамках процесса, предназначенные для загрузки компоновочных блоков .NET. Домен приложения, в свою очередь, можно делить дальше на контекстные области со своими границами. В сущности, контекст .NET обеспечивает домену приложения возможность создать "Отдельную комнату" для данного объекта.

Используя контекст, среда CLR может гарантировать, что объекты, которые выдвигают специальные требования в среде выполнении, будут обработаны надлежащим образам и в нужном порядке, поскольку для этого среда использует перехват вызовов, пересекающих границу контекста. Слой перехвата позволяет среде CLR корректировать текущие вызовы методов в соответствии с контекстно-зависимыми установками данного объекта. Например, если вы определите тип класса C#, требующий автоматической поддержки множества потоков (используя атрибут [Synchronization]), то среда CLR при его размещении создаст "синхронизированный контекст".

Точно так же, как процесс определяет создаваемый по умолчанию домен приложения, каждый домен приложения имеет создаваемый по умолчанию контекст. Этот создаваемый по умолчанию контекст (на который иногда ссылаются, как на *контекст 0,*поскольку он всегда оказывается первым контекстом, создаваемым в домене приложений) применяется для тех объектов .NET, которые не имеют никаких особых или уникальных контекстных требований. Вы вправе ожидать, что подавляющее большинство объектов .NET должно загружаться в контекст 0. Когда среда CLR определяет новый объект, имеющий специальные требования, создаются новые границы контекста в рамках объемлющего домена приложения. На рис. 13.9 показана схема взаимодействия процесса, доменов приложения и контекстов.

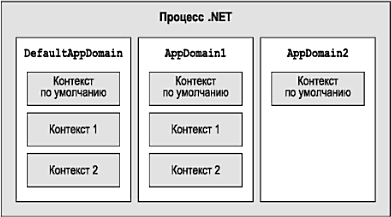


Рис. 13.9. Процессы, домены приложения и границы контекста

**Контекстно-независимые и контекстно-связанные типы**

Типы .NET которые не предъявляют никаких специальных контекстных требований, называются *контекстно-независимыми*объектами. Эти объекты доступны из любого места в рамках соответствующего домена приложения, без каких бы то ни было особых требований к среде выполнения. Построение контекстно-независимых объектов не требует больших усилий, поскольку ничего специального делать не приходится (в частности, не требуется наделять тип контекстными атрибутами или получать производные базового класса System.ContextBoundObject).

**// Контекстно-независимый объект загружается в контекст 0.**

public class SportsCar()

С другой стороны, объекты, которые требуют контекстного размещения, называются *контекстно-связанными*объектами, и они должны быть производными от базового класса System.ContextBoundObject. Этот базовый класс закрепляет тот факт, что соответствующий объект сможет правильно функционировать только в рамках контекста, в котором он был создан. С учетом роли контекста .NET должно быть ясно, что в случае, когда контекстно-связанный объект оказывается в несоответствующем контексте, в любой момент могут возникнуть проблемы.

Вдобавок к необходимости получения типа из System.ContextBoundObject, контекстно-связанный тип будет наделен рядом специальных атрибутов .NET, называемых *контекстными атрибутами*(что вполне логично). Все контекстные атрибуты получаются из базового класса. System.Runtime.Remoting.Contexts. ContextAttribute:

public class System.Runtime.Remoting.Contexts.ContextAttribute: Attribute, IContextAttribute, IContextProperty {

 public ContextAttribute(string name);

 public string Name { virtual get; }

 public object TypeId { virtual get; }

 public virtual bool Equals(object o);

 public virtual void Freeze(System.Runtime.Remoting.Contexts.Context newContext);

 public virtual int GetHashCode();

 public virtual void GetPropertiesForNewContext(System.Runtime.Remoting.Activation.IConstructionCallMessage сtorMsg);

 public Type GetType();

 public virtual bool IsContextOK(System.Runtime.Remoting.Contexts.Context ctx, System.Runtime.Remoting.Activation.IConstructionCallMessage ctorMsg);

 public virtual bool IsDefaultAttribute();

 public virtual bool IsNewContextOK(System.Runtime.Remoting.Contexts.Context newCtx);

 public virtual bool Match(object obj);

 public virtual string ToString();

}

Поскольку класс ContextAttribute не является изолированным, вполне возможно строить свои собственные пользовательские контекстные атрибуты (для этого следует получить класс, производный от ContextAttribute, и переопределить необходимые виртуальные методы). После этого вы сможете создать пользовательский программный код, отвечающий на контекстные установки.

**Замечание.**В этой книге не рассматриваются подробности создания пользовательских контекстов объектов, но если вы заинтересованы узнать об этом больше, прочитайте книгу *Applied .NET Attributes*(Apress, 2003).

**Определение контекстно-связанных объектов**

Чтобы определить класс (SportsCarTS), автоматически поддерживающий потоковую безопасность, без добавления в него сложной логики синхронизации патока при реализации членов, следует взять объект, производный от ContextBoundObject, и применить атрибут [Synchronization], как показано ниже.

using System.Runtime.Remoting.Contexts;

**// Этот контекстно-связанный тип будет загружен только**

**// в синхронизированном (т.е. многопоточном) контексте.**

**[Synсhronization]**

public class SportsCarTS: ContextBoundObject{}

Типы с атрибутом [Synchronization] загружаются в контексте сохранения потоков. С учетом специальных контекстуальных требований типа класса MyThreadSafeObject представьте себе те проблемы, которые должны возникнуть, если размещенный объект перевести из синхронизированного контекста в несинхронизированный. Объект вдруг перестанет быть защищенным в отношении потоков и превратится в потенциального нарушителя целостности данных, поскольку другие потоки могут пытаться взаимодействовать с этим ссылочным объектом (теперь уже не сохраняющим потоки). Для гарантии того, что среда CLR не переместит объекты SportsCarTS за рамки синхронизированного контекста, достаточно взять объект, производный от ContextBoundObject.

**Проверка контекста объекта**

Из тех приложений, которые вы построите сами, очень немногие могут потребовать программного взаимодействия с контекстом, но вот вам пример для иллюстрации подхода, о котором идет речь. Создайте новое консольное приложение с именем ContextManipulator. Это приложение будет определить один контекстно-независимый класс (SportsCar) и один контекстно-связанный (SportsCarTS).

using System.Runtime.Remoting.Contexts;**// Для типа Context.**

using System.Threading; **// Для типа Thread.**

**// Тип SportsCar не имеет специальных контекстных требований**

**// и будет загружен в рамках контекста, создаваемого доменом**

**// приложения по умолчанию.**

public class SportsCar {

 public SportsCar() {

**// Чтение информации и вывод идентификатора контекста.**

  Context ctx = Thread.CurrentContext;

  Console.WriteLine("{0} объект в контексте {1}", this.ToString(), ctx.ContextID);

  foreach (IContextProperty itfCtxProp in ctx**.ContextProperties)**Console.WriteLine("-› Свойство контекста: {0}", itfCtxProp.Name);

 }

}

**// Тип SportsCarTS требует загрузки**

**// в синхронизированном контексте.**

[Synchronization]

public class SportsCarTS: ContextBoundObject {

 public SportsCarTS() {

**// Чтение информации и вывод идентификатора контекста.**

  Context ctx = Thread.CurrentContext;

  Console.WriteLine("{0} объект в контексте {1}", this.ToString(), ctx.ContextID);

  foreach(IContextProperty itfCtxProp in ctx.ContextProperties) Console.WriteLine("-› Свойство контекста: {0}", itfCtxProp.Name);

 }

}

Обратите внимание на то. что каждый конструктор получает тип Context от текущего потока выполнения через статическое свойство Thread.CurrentContext. Используя объект Context, вы можете распечатать информацию о границах контекста, например, значение ID контекста или значения дескрипторов, полученных через Context.ContextProperties. Это свойство возвращает объект, реализующий интерфейс IContextProperty, который обеспечивает доступ к дескрипторам с помощью свойства Name. Теперь обновите метод Main(), чтобы разместить по экземпляру каждого из типов класса.

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Чудесное контекстное приложение \*\*\*\n");

**// При создании объекты будут отображать информацию контекста.**

 SportsCar sport = new SportsCar();

 Console.WriteLine();

 SportsCar sport2 = new SportsCar();

 Console.WriteLine();

 SportsCarTS synchroSport = new SportsCarTS();

 Console.ReadLine();

}

По мере создания объектов конструкторы классов отображают различные элементы информации о контексте (рис. 13.10).

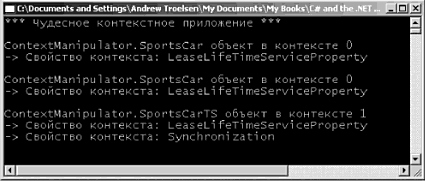


Рис. 13.10. Исследование контекста объекта

Для класса SportsCar не был указан атрибут контекста, поэтому среда CLR размещает sport и sport2 в контексте 0 (т.е. в контексте, созданном по умолчанию). Однако объект SportsCarTS загружается в свои уникальные контекстуальные границы (которым назначается идентификатор 1), поскольку для этого контекстно-связанного типа был указан атрибут [Synchronization].

**Исходный код**. Проект ContextManipulator размещен в подкаталоге, соответствующем главе 13.

**Еще несколько слов о процессах, доменах приложения и контекстах**

К этому моменту вы должны лучше понимать, как среда CLR обрабатывает компоновочные блоки .NET. Вот на что следует обратить внимание.

• Процесс .NET может содержать один или несколько доменов приложения. Каждый домен приложения может принять любое число связанных компоновочных блоков .NET и независимо загружаться и выгружаться средой CLR (или программистом с помощью типа System.AppDomain).

• Любой домен приложения состоит из одного или нескольких контекстов. Используя контексты, среда CLR может поместить объект со "специальными требованиями" в логический контейнер, чтобы гарантировать выполнение этих требований в среде выполнения.

Если предыдущее обсуждение кажется вам слишком сложным и далеким от практики, не волнуйтесь. По большей части среда выполнения .NET автоматически разрешает вопросы процессов, доменов приложений и контекстов, не требуя вашего вмешательства. Тем не менее представленная здесь информация обеспечивает "твердую основу" для понимания принципов многопоточного программирования в рамках платформы .NET. Но перед тем, как перейти к изучению пространства имен System.Threading, мы попытаемся выяснить, как сама среда CLR обрабатывается операционной системой Win32.

**Хостинг общеязыковой среды выполнения**

Для конечного пользователя запуск выполняемого блока .NET доступен с помощью простого двойного щелчка на соответствующем файле \*.exe в окне программы Проводник (или активизации соответствующего ярлыка). Но вы должны помнить из главы 1, что каркас .NET Framework (пока что) не интегрирован непосредственно в ОС Windows, а опирается на ОС. Во время установки Visual Studio 2005 (или .NET Framework 2.0 SDK) на вашу машину устанавливается и окружение среды выполнения .NET (включая все необходимые библиотеки базовых классов). Также напомним, что Microsoft предлагает свободно доступную программу установки (dotnetfx.exe) среды выполнения .NET, позволяющую настроить машину конечного пользователя на поддержку компоновочных блоков .NET.

Поскольку ОС Windows не имеет встроенных средств понимания формата компоновочных блоков .NET, полезно знать, что происходит в фоновом режиме, когда активизируется выполняемый компоновочный блок. В ОС Windows XP основными шагами будут следующие (вспомните из главы 11, что все компоновочные блоки .NET содержат информацию заголовка Win32).

1. ОС Windows загружает выполняемый двоичный файл в память.

2. ОС Windows читает встроенный заголовок WinNT, чтобы определить (по флагу IMAGE\_DIRECTORY\_ENTRY\_COM\_DESCRIPTOR), является ли двоичный файл компоновочным блоком .NET.

3. Если образ является компоновочным блоком .NET, загружается mscoree.dll.

4. Затем mscoree.dll загружает одну *из двух*реализаций CLR (mscorwks.dll или mscorsvr.dll).

5. В этот момент ответственность за выполнение "принимает на себя" среда CLR, выполняющая все связанные с .NET задачи (поиск внешних компоновочных блоков, выполнение проверок безопасности, обработка CIL-кода, сборка мусора и т.д.).

Итак, mscoree.dll - это не сама CLR (как говорилось в предыдущих главах). Хотя вполне возможно идентифицировать mscoree.dll с реальной CLR, на самом деле указанный двоичный файл – это "развилка" на пути к одной из двух возможных реализаций CLR. Если соответствующая машина использует один процессор, загружается mscorwks.dll. Если машина поддерживает мультипроцессорный режим, в память загружается mscorsvr.dll (это версия CLR, оптимизированная для работы на машинах с несколькими процессорами).

**Параллельное выполнение CLR**

"Копнув" чуть глубже, мы увидим, что платформа .NET поддерживает параллельное выполнение, т.е. на одной машине можно установить несколько версий платформы .NET (во время создания этой книги были доступны версии 1.0.1.1 и 2.0). Сам файл mscoree.dll размещается в подкаталоге System32 каталога установки Windows. Например, на моей машине mscoree.dll "проживает" в каталоге C:\WINDOWS\system32 (рис. 13.11).

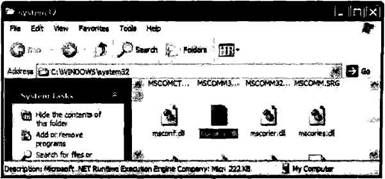


Рис. 13.11. Файл mscoree.dll находится в каталоге system32

После загрузки mscoree.dll по реестру системы Win32 (да, по реестру этой системы) выясняется номер последней из установленных версий и путь установки .NET Framework (используется ветвь HKEY\_LOCAL\_MACHINE\Software\Microsoft\.NETFramework, рис. 13.12).

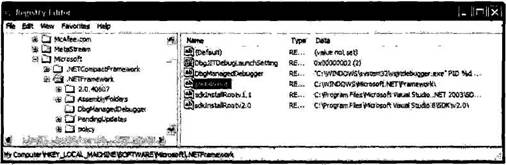


Рис. 13.12. Выяснение версии и пути установки платформы .NET

После определения версии и пути установки платформы .NET в память загружается нужная версия mscorwks.dll/mscorsvr.dll. На моей машине корневым путем установки платформы .NET является C:\WINDOWS\Microsoft.NET\Frаmеwork. В указанном каталоге есть специальные подкаталоги для .NET версии 1.0.1.1 и (на время создания книги) текущей версии 2.0 (см. рис. 13.13, ваши номера версий могут быть другими).

**Загрузка конкретной версии CLR**

Когда mscoree.dll определяет (с помощью реестра системы), какую версию mscorwks.dll/mscorsrv.dll загрузить, читается также раздел Policy (Политика) ветви HKEY\_LOCAL\_MACHINE\Software\Microsoft\.NETFramework реестра. В этот раздел записывается информация обновлений CLR, которые могут выполняться с безопасностью. Например, если запускается компоновочный блок, который был построен с использованием *.*NET версии 1.0.3.705, mscoree.dll узнает из файла политики, что вполне безопасно загрузить версию 1.1.4322.

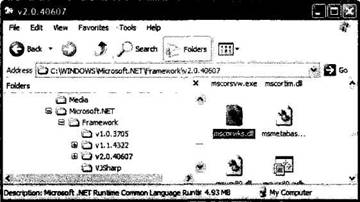


Рис. 13.13. Файл mscorwks.dll версии 2.0

Все это происходит незаметно в фоновом режиме и только тогда, когда известно, что обновление обеспечивает правильное выполнение. В редких случаях возникает необходимость заставить mscoree.dll загрузить конкретную версию CLR, и тогда вы можете использовать для этого файл \*.config клиента.

‹?xml version="l.0" encoding="utf-8"?›

 ‹configuration›

  ‹startup›

**‹requiredRuntime version ="1.0 .3705"/›**

  ‹/startup›

‹/configuration›

Здесь элемент ‹requiredRuntime› указывает, что для загрузки данного компоновочного блока следует использовать только версию 1.0.3705. Поэтому, если на целевой машине нет полной инсталляции .NET версии 1.0.3705, конечный пользователь увидит окно с информацией об ошибке среды выполнения, показанное на рис. 13.14.



Рис. 13.14. Элемент ‹requiredRuntime› порождает сообщение об ошибке среды выполнения, если указанная версия CLR не установлена[[2]](http://lib.rus.ec/b/167642/read" \l "n_2" \o "    Сообщение гласит: \"Чтобы выполнить это приложение, установите одну из следующих версий .NET Framework: v1.0.3705. Обратитесь к разработчику приложения для получения инструкций по поводу установки нужной версии .NET Framework.\"   )

**Дополнительные хосты CLR**

Только что описанный процесс обозначил основные шаги, предпринимаемые операционной системой Windows для хостинга CLR по умолчанию, когда запускается выполняемый компоновочный блок. Но Microsoft предлагает множество приложений, которые могут действовать в обход используемого по умолчанию поведения, используя *программную*загрузку CLR. Например. Microsoft Internet Explorer может загружать своими встроенными средствами пользовательские элементы управления Windows Forms (управляемый эквивалент теперь уже устаревших элементов управления ActiveX). Последняя версия Microsoft SQL Server (с кодовым названием Yukon и официальным названием SQL Server 2005) также способна осуществлять непосредственный хостинг CLR.

Наконец. Microsoft определила набор интерфейсов, позволяющих разработчикам строить их собственные пользовательские хосты CLR. Это можно сделать*,*используя соответствующий программный код C/C++ или библиотеку COM-типа (mscorеe.tlb). Хотя сам процесс построения пользовательского хоста CLR исключительно прост (особенно при использовании библиотеки COM-типа), эта тема выходит за рамки нашего обсуждения. Если вам нужна дополнительная информация по данному вопросу, в Сети вы можете найти множество статей на эту тему (просто выполните поиск по ключу "CLR hosts").

**Резюме**

Целью этой главы было выяснение того, как обрабатывается выполняемый образ .NET. Вы имели возможность убедиться в том, что уже привычное понятие процесса Win32 было внутренне изменено с тем, чтобы адаптировать его к требованиям CLR. Отдельный процесс (которым можно программно управлять с помощью типа System.Diagnostiсs.Process) теперь компонуется из множества доменов приложения, имеющих изолированные и независимые границы в рамках этого процесса. Один процесс может содержать множество доменов приложения, каждый из которых может обрабатывать и выполнять любое число связанных компоновочных блоков.

Кроме того, каждый домен приложения может содержать любое число контекстов. Используя этот дополнительный уровень изоляции типов, среда CLR может гарантировать, что объекты со специальными требованиями будут обработаны корректно. Глава завершается рассмотрением деталей того, как сама CLR обрабатывается операционной системой Win32.

**ГЛАВА 14. Создание многопоточных приложений**

В  предыдущей главе мы рассмотрели взаимосвязь между процессами, доменами приложения и контекстами. В этой мы выясним, как в рамках платформы .NET строить многопоточные приложения и как в условиях множества потоков гарантировать целостность совместно используемых ресурсов.

Наше обсуждение снова начнется с рассмотрении типа делегата .NET, чтобы прийти к пониманию его внутренней поддержки асинхронных вызовов методов. Вы увидите, что такой подход позволяет автоматически вызвать метод во вторичном потоке выполнения. Затем мы исследуем типы пространства имен System.Тhreading. Будет рассмотрено множество типов (Thread.ThreadStart и т.д.), позволяющих с легкостью создавать дополнительные потоки. Конечно, сложность разработки многопоточных приложений заключается не в создании потоков, а в гарантии того, что ваш программный код будет иметь надежные средства обработки конфликтов при конкурентном доступе к общедоступным ресурсам. Поэтому завершается глава рассмотрением различных примитивов синхронизации, предлагаемых каркасом .NET Framework.

**Взаимосвязь процессов, доменов приложений, контекстов и потоков**

В предыдущей главе обсуждалось понятие *потока,*который был определен, как путь исполнения в рамках выполняемого приложения. И хотя многие приложения .NET имеют только один поток и, тем не менее, оказываются очень полезными, первичный поток компоновочного блока (порождаемый средой CLR при выполнении Main()) может создавать вторичные потоки для решения дополнительных задач. Реализуя дополнительные потоки, вы можете строить приложения с лучшим откликом на действия пользователя (но не обязательно более быстро выполняющие свои задачи).

Пространство имен System.Threading содержит различные типы, позволяющие создавать многопоточные приложения. Основным типом здесь можно считать класс Thread, поскольку он представляет данный поток. Чтобы программно получить ссылку на поток, выполняющий данный член в текущий момент, просто вызовите статическое свойство Thread.CurrentThread.

private static void ExtractExecutingThread() {

**// Получение потока, выполняющего**

**// в данный момент данный метод.**

Thread currThread = Thread.CurrentThread;

}

Для платформы .NET *не предполагается*прямого однозначного соответствия между доменами приложения и потоками. Напротив, домен приложения может иметь множество потоков, выполняющихся в рамках этого домена в любой момент времени. Кроме того, конкретный поток не привязан к одному домену приложения в течение всего времени существования потока. Потоки могут пересекать границы домена приложения, подчиняясь правилам потоков Win32 и целесообразности CLR.

Но, хотя активные потоки могут перемещаться через границы доменов приложения, в любой конкретный момент времени один конкретный поток может выполняться в рамках только одного домена приложения (другими словами, один поток не может работать в нескольких доменах приложения одновременно). Чтобы программно получить доступ к домену приложения, содержащему текущий поток, следует вызвать статический метод Thread.GetDomain().

private static void ExtractAppDomainHostingThread() {

**// Получение домена приложения, содержащего текущий поток.**

 AppDomain ad = Thread.GetDomain();

}

Любой поток в любой момент времени также может быть перемещен средой CLR в любой из имеющихся контекстов или помещен в новый контекст. Чтобы получить текущий контекст, в рамках которого оказался поток, используйте статическое свойство Thread.CurrentContext.

private static void ExtractCurrentThreadContext() {

**// Получение контекста, в рамках которого**

**// действует текущий поток.**

Context ctx = Thread.CurrentContext;

}

Снова подчеркнем, что именно среда CLR является тем объектом, который отвечает за помещение потоков в соответствующие домены приложения и контексты. Как разработчик приложений .NET, вы обычно остаетесь в блаженном неведении относительно того, где заканчивается данный поток (или, точнее, когда он помещается в новые границы). Однако вам будет полезно знать различные способы получения соответствующих примитивов.

**Проблема конкуренции и роль синхронизации потоков**

Одним из множества "преимуществ" (читайте источников проблем) многопоточного программирования является то, что вы имеете очень узкие возможности контроля в отношении использования потоков операционной системой и средой CLR. Например, построив блок программного кода, создающий новый поток выполнения, вы не можете гарантировать, что этот поток начнет выполняться немедленно. Скорее, такой программный код только "даст инструкцию" операционной системе начать выполнение потока как можно быстрее (что обычно означает момент, когда наступит очередь этого потока у планировщика потоков).

Кроме того, поскольку потоки могут перемещаться между границами приложения и контекста по требованию CLR, вы должны, следить за тем, какие элементы вашего приложения *открыты влиянию потоков* (т.е. позволяют доступ множества потоков), а какие операции оказываются *атомарными*(операции, открытые для множества потоков, потенциально опасны!). Для примера предположим, что поток вызывает некоторый метод конкретного объекта. Предположим также, что после этого поток, получает инструкцию от планировщика потоков приостановить выполнение, чтобы позволить другому потоку доступ к тому же методу того же объекта.

Если оригинальный поток еще не завершил свою текущую операцию, второй входящий поток может получить дли просмотра объект в частично измененном состоянии. В этом случае второй поток, по сути, будет читать некорректные данные, в результате чего возникнут досадные (и очень трудные для выявления) ошибки, которые характеризуются неустойчивостью при воспроизведении и отладке.

Атомарные операции, с другой стороны, всегда безопасны в многопоточном окружении. К сожалению, только для очень небольшого числа операций из библиотек базовых классов .NET можно гарантировать, что эти операции будут атомарными. Не является атомарной даже операция присваивание значения члену-переменной! Если в документации .NET Framework 2.0 SDK в отношении какой-либо операции специально не оговорено, что данная операция является атомарной, вы должны предполагать, что эта операция является открытой влиянию потоков и принимать специальные меры предосторожности.

Теперь вам должно быть ясно, что домены многопоточного приложения тоже открыты влиянию потоков, поскольку потоки могут пытаться использовать доступные функциональные возможности одновременно. Чтобы защитить ресурсы приложения от возможных искажении, разработчикам .NET приходится использовать так называемые примитивы потоков (такие, как блокировки, мониторы и атрибут [Synchronization]), чтобы контролировать доступ выполняемых потоков.

Нельзя утверждать, что платформа .NET исключила все трудности, возникающие при построении устойчивых многопоточных приложений, но теперь этот процесс значительно упрощён. Используя типы, определенные в пространстве имен System.Threading, вы получаете возможность создавать дополнительные потоки с минимальными усилиями и минимальными проблемами. Точно так же, когда приходит время блокировать открытые элементы данных, вы можете использовать дополнительные типы, которые обеспечивают те же функциональные возможности, что и примитивы потоков Win32 API (но при этом используется намного более аккуратная объектная модель).

Однако использование пространства имея System.Threading – это не единственный путь построения многопоточных программ .NET. В ходе нашего обсуждения делегатов (см. главу 8) мы уже упоминали о том, что все делегаты NET обладают способностью асинхронного вызова членов. Это – *главное*преимущество платформы .NET, поскольку одной из основных причин, в силу которых разработчик создает потоки, является необходимость такого вызова методов, при котором не возникает блокировок (т.е. именно асинхронного вызова). Для достижения такого результата можно использовать и пространство имен System.Threading, но с помощью делегатов это делается намного проще.

**Краткий обзор делегатов .NET**

Напомним, что тип делегата .NET – это обеспечивающий типовую безопасность объектно-ориентированный указатель функции. Когда вы объявляете делегат .NET, компилятор C# отвечает на это созданием изолированного класса, полученного из System.MulticastDelegate (который, в свою очередь, является производным от System.Delegate). Эти базовые классы наделяют каждый делегат способностью поддерживать список адресов методов, которые могут быть вызваны позднее. Давайте рассмотрим декларацию делегата BinaryOp, который был впервые определен в главе 8.

**// Тип делегата C#.**

public delegate int BinaryOp(int x, int y);

В соответствии с данным определением BinaryOp может указывать на любой метод с двумя целочисленными аргументами, возвращающий целочисленное значение. После компиляции соответствующий компоновочный блок будет содержать полноценное определение класса, которое динамически генерируется на основе декларации делегата. В случае BinaryOp это определение класса будет выглядеть приблизительно так (приводится в псевдокоде).

sealed class BinaryOp: System.MulticastDelegate {

 public **BinaryOp**(object target, uint functionAddress);

 public **void**Invoke**(int**x, **int**y);

 public IAsyncResult BeginInvoke**(int**x, **int**y, AsyncCallback cb, object state);

 public **int**EndInvoke(IAsyncResult result);

}

Напомним, что генерируемый метод Invoke() используется для вызова методов, обслуживаемых объектом делегата в синхронном режиме. В этом случае вызывающий поток (например, первичный поток приложения) вынужден ждать, пока не завершится вызов делегата. Также напомним, что в C# метод Invoke() не вызывается в программном коде явно, а запускается в фоновом режиме при использовании "нормального синтаксиса" вызова метода. Рассмотрите следующий программный код, в котором статический метод Add() вызывается в синхронной (т.е. блокирующей) форме.

**// Это требуется для вызова Thread.Sleep().**

using System.Threading;

using System;

namespace SyncDelegate {

 public delegate int BinaryOp(int x, int y);

 class Program {

  static void Main(string[] args) {

   Console.WriteLine("\*\*\*\*\* Синхронный вызов, делегата \*\*\*\*\*");

***//*Вывод ID выполняемого потока.**

   Console.WriteLine("Вызван Main() в потоке {0}.", Thread.CurrentThread.GetHashCode());

**// Вызов Add() в синхронной форме.**

   BinaryOp b = new BinaryOp(Add);

   int answer = b(10, 10);

**// Эти строки не будут выполнены до завершения**

**// работы метода Add().**

   Console.WriteLine("В Main() еще есть работа!");

   Console.WriteLine("10 + 10 равно {0}.", answer);

   Console.ReadLine();

  }

  static int Add(int *x,*int y) {

**// Вывод ID выполняемого потока.**

Console.WriteLine("Вызван Add() в потоке {0}.", Thread.CurrentThread.GetHashCode());

**// Пауза примерно 5 секунд для**

**// имитации длительной операции.**

   Thread.Sleep(5000);

   return x + у;

  }

 }

}

Сначала заметим, что в этой программе используется пространство имен System.Threading. В методе Add() вызывается статический метод Thread.Sleep(), чтобы приостановить вызывающий поток (приблизительно) на пять секунд для имитации задачи, выполнение которой требует много времени. Поскольку метод Add() вызывается в *синхронной*форме, метод Main() не напечатает результат операции до тех пор, пока не завершится работа метода Add().

Далее заметим, что метод Main() получает доступ к текущему потоку (с помощью Thread.CurrentThread) и печатает его хешированный код. Поскольку этот хешированный код представляет объект в конкретном состоянии, соответствующее значение можно использовать как "грубый" идентификатор потока. Та же логика используется в статическом методе Add()*.*Как и следует ожидать, поскольку вся работа в этом приложении выполняется исключительно первичным потоком, вы увидите одинаковые хешированные значения в консольном выводе программы (рис. 14.1).

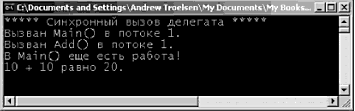


Рис. 14.1. Синхронные вызовы методов "блокируют" другие вызовы

При выполнении этой программы вы заметите, что перед тем выполнением Console.WriteLine() произойдет пятисекундная задержка. И хотя многие методы (если не подавляющее их большинство) могут вызваться синхронно совершенно безболезненно, делегатам .NET, если это необходимо, можно дать указание вызывать методы асинхронно.

**Исходный код.**Проект SyncDelegate размещен в подкаталоге, соответствующем главе 14.

**Асинхронная природа делегатов**

Если для вас тема многопоточных приложений является новой, вы можете спросить, чем же на самом деле является асинхронный вызов метода. Вы, без сомнения, знаете о том, что для выполнения некоторых программных операций требуется время. Предыдущий метод Add() был исключительно иллюстративным, но представьте себе, что вы построили однопоточное приложение, в котором вызывается метод удаленного объекта, выполняющий сложный запрос к большой базе данных или запись 500 строк текста во внешний файл. Пока не закончится выполнение этих операций, приложение будет казаться зависшим достаточно долгое время. Пока соответствующая задача не будет обработана, все другие возможности программы (такие как, например, активизация меню, выбор элементов в панели инструментов или вывод на консоль) будут недоступны для пользователя.

Но как дать указание делегату вызвать метод в отдельном потоке выполнения, чтобы имитировать одновременное выполнение множества задач? К счастью, нужной для этого способностью автоматически наделяется каждый тип делегата .NET. И более того, для такого вызова вам *не требуется*углубляться в детали пространства имен System.Threading (хотя, естественно, одно другому не мешает).

**Методы BeginInvoke() и EndInvoke()**

Когда компилятор C# обрабатывает ключевое слово delegate, динамически генерируемый класс определяет два метода с именами BeginInvoke() и EndInvoke(). Для нашего определения делегата BinaryOp эти методы оказываются следующими.

sealed class **BinaryOp**: System.MulticastDelegate {

 …

**// Используется для асинхронного вызова метода.**

 public IAsyncResult BeginInvoke(int x*,*int y, AsyncCallback cb, object state);

**// Используется для извлечения возвращаемого значения**

**// вызванного метода.**

 public int EndInvoke(IAsyncResult result);

}

Первый набор параметров, передаваемых в BeginInvoke(), формируется на основе формата делегата C# (в случае BinaryOp это два целочисленных значения). Последними двумя аргументами всегда являются System.AsyncCallback и System.Object. Мы рассмотрим роль этих параметров чуть позже, а пока что для каждого из них мы будем использовать null.

**Интерфейс System.IAsyncResult**

Метод BeginInvoke() всегда возвращает объект, реализующий интерфейс IAsyncResult, а метод EndInvoke() имеет единственный параметр типа IAsyncResult. Совместимый с IAsyncResult объект, возвращаемый методом BeginInvoke(), и является тем связующим механизмом, который позволяет вызывающему потоку получить результат асинхронного вызова метода позже с помощью EndInvoke(). Интерфейс IAsyncResult (определенный в пространстве имен System) задается так, как показано ниже.

public interface **IAsyncResult**{

 object AsyncState { get; }

 WaitHandle AsyncWaitHandle { get; }

 bool CompletedSynchronously { get; }

 bool IsCompleted { get; }

}

В самом простом случае можно избежать непосредственного вызова этих членов. Требуется только сохранить совместимый с IAsyncResult объект, возвращенный BeginInvoke(), и передать его методу EndInvoke(), когда вы будете готовы получить результат вызова метода. Позже вы увидите, что у вас есть возможность вызывать члены совместимого с IAsyncResult объекта, если вы хотите "участвовать" в процессе извлечения возвращаемого значений метода.

**Замечание.**Если асинхронно вызывается метод, который не предлагает возвращаемых значений, можно его вызвать и престо "забыть" о нем. В таких случаях нет необходимости сохранять совместимый с IAsyncResult объект и вызывать EndInvoke() (так как нет возвращаемого значения, которое требуется извлечь).

**Асинхронный вызов методов**

Чтобы дать указание делегату BinaryOp вызвать метод Add() асинхронно, измените предыдущий метод Main() так, как показано ниже.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* асинхронный вызов делегата \*\*\*\*\*");

**// Вывод ID выполняемого потока.**

 Console.WriteLine("Вызван Main() в потоке {0}.", Thread.CurrentThread.GetHashCode());

**// Вызов Add() во вторичном потоке.**

 BinaryOp b = new BinaryOp(Add);

 IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

**// Выполнение другой работы в первичном потоке.…**

 Console.WriteLine("В Main() еще есть работа!");

**// Получение результата метода Add(),**

**// когда это требуется.**

 int answer = b.EndInvoke(iftAR);

 Console.WriteLine ("10 + 10 равно {0}.", answer);

 Console.ReadLine();

}

Выполнив это приложение, вы увидите, что теперь выводятся два разных хешированных значения, поскольку в границах текущего домена приложения выполняются два потока (см. рис. 14.2).



Рис. 14.2. Методы, вызываемые асинхронно, выполняют свою работу в отдельном потоке

Вдобавок к уникальным хешированным значениям, вы также обнаружите, что при запуске приложения сообщение "В Main() еще есть работа!" появляется практически немедленно.

**Синхронизация вызывающего потока**

Для текущей реализации Main() диапазон времени между вызовом BeginInvoke() и вызовом EndInvoke() явно меньше пяти секунд. Поэтому после вывода на консоль сообщения "В Main() еще есть работа!" поток вызова блокируется и ждет завершения существования вторичного потока, который должен получить результат метода Add(). Таким образом, вы на самом деле выполняете еще один *синхронный вызов.*

static void Main (string[] args) {

 …

 BinaryOp b = new BinaryOp(Add);

 IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

**// До этого вызова проходит менее 5 секунд!**

 Console.WriteLine("В Main() еще есть работа!");

**// Вызывающий поток блокируется до завершения EndInvoke().**

 int answer = b.EndInvoke(iftAR);

 …

}

Очевидно, что асинхронные делегаты теряют свою привлекательность, если поток вызова может при определенных условиях блокироваться. Чтобы позволить вызывающему потоку выяснить, закончил ли асинхронно вызванный метод свою работу, интерфейс IAsyncResult предлагает свойство IsCompleted. Используя этот член, поток вызова может перед вызовом EndInvoke() проверить, завершен ли асинхронный вызов. Если работа метода не завершена, IsCompleted возвращает false (ложь), и поток вызова может продолжать свою работу. Если же IsCompleted возвращает true (истина), то поток вызова может получить результат "наименее блокирующим" способом. Рассмотрите следующую модификацию метода Main().

static void Main (string[] args) {

 …

 BinaryOp b = new BinaryOp(Add);

 IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

**// Это сообщение будет печататься до тех пор, // пока не завершится вызов метода Add().**

 while (!iftAR.isCompleted) Console. WriteLine("В Main() еще есть работа!");

**// Теперь мы знаем, что вызов метода Add() завершен.**

 int answer = b.EndInvoke(iftAR);

 …

}

Здесь вводится цикл, который будет продолжать выполнение оператора Console.WriteLine() до тех пор, пока не завершится вторичный поток. Как только это произойдет, вы сможете получить результат метода Add() с уверенностью, что этот метод завершил свою работу.

Вдобавок к свойству IsCompleted интерфейс IAsyncResult предлагает свойство AsyncWaitHandle для построения еще более гибкой логики ожидания. Это свойство возвращает экземпляр WaitHandle, предлагающий метод WaitOne(). Преимущество метода WaitHandle.WaitOne() в том, что вы можете указать максимальное время ожидания. Если указанное время превышено, WaitOne() возвращает false. Рассмотрите следующий (обновленный) вариант цикла while:

while (!iftAR.AsyncWaitHandle.WaitOne(2000, true)) {

 Console.WriteLine("В Main() еще есть работа!");

}

Указанные свойства IAsyncResult и в самом деле обеспечивают возможность синхронизации потока вызова, но этот подход оказывается не самым эффективным. Во многих отношениях свойство IsCompleted подобно назойливому менеджеру (или однокласснику), который постоянно спрашивает: "Уже все сделал?" К счастью, делегаты предлагают целый ряд других (и более действенных) подходов для получения результатов методов, вызываемых асинхронно.

**Исходный код.**Проект AsyncDelegate размещен в подкаталоге, соответствующем главе 14.

**Роль делегата AsyncCallback**

Вместо того чтобы выяснять у делегата, завершился ли асинхронный вызов метода, лучше позволить делегату информировать поток вызова о выполнении задания. Чтобы реализовать такое поведение, вы должны предъявить экземпляр делегата System.AsyncCallback методу BeginInvoke() в виде параметра, значением которого до сих пор было у нас значение null. Если вы укажете AsyncCallback, делегат вызовет соответствующий метод автоматически, когда асинхронный вызов завершится.

Подобно любому другому делегату, AsyncCallback может вызывать только методы, соответствующие конкретному шаблону, и в данном случае это методы, принимающие единственный параметр типа IAsyncResult и возвращающие void.

void MyAsyncCallbackMethod(IAsyncResult iftAR)

Предположим, что у нас есть другое приложение, использующее делегат BinaryOp. На этот раз мы не будем "просить" делегат выяснить, завершился ли метод Add(). Вместо этого мы определим статический метод с именем AddComplete(), чтобы получить извещение о завершении асинхронного вызова,

namespace AsyncCallbackDelegate {

 public delegate int BinaryOp(int x, int y);

 class Program {

  static void Main(string[] args) {

   Console.WriteLine("\*\*\* Пример делегата AsyncCallback \*\*\*");

   Console.WriteLine("Вызван Main() в потоке {0}", Thread.CurrentThread.GetHashCode());

   BinaryOp b = new BinaryOp(Add);

   IAsyncResult iftAR = b.BeginInvoke(10, 10, **new AsyncCallback(AddComplete)**, null);

**// Здесь выполняется другая работа…**

   Console.ReadLine();

  }

  static void AddComplete(IAsyncResult iftAR) {

   Console.WriteLine("Вызван AddComplete() в потоке {0}", Thread.CurrentThread.GetHashCode());

   Console.WriteLine("Ваше сложение выполнено");

  }

  static int Add(int x, int y) {

   Console.WriteLine("Вызван Add() в потоке {0}.", Thread.CurrentThread.GetHashCode());

   Thread.Sleep(5000);

   return x + y;

  }

 }

}

Снова заметим, что статический метод AddComplete() будет вызван делегатом AsyncCallback тогда, когда завершится вызов метода Add(). Выполнение этой программы может подтвердить, что именно вторичный поток выполняет обратный вызов AddComplete() (рис. 14.3).

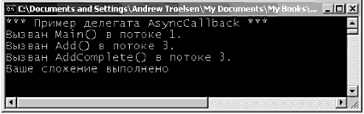


Рис. 14.3. Делегат *AsyncCallback*в действии

**Роль класса AsyncResult**

В текущей своей форме метод Main() не хранит тип IAsyncResult, возвращаемый из BeginInvoke(), и не вызывает EndInvoke(). Более того, целевой метод делегата AsyncCallback (в данном случае это метод AddComplete()) вообще не имеет доступа к оригинальному делегату BinaryOp, созданному в контексте Main(). Можно, конечно, объявить BinaryOp, как статический член класса, чтобы позволить обоим методам иметь доступ к объекту, но более "элегантным" решением яв-ляетcя использование входного параметра IAsyncResult.

Поступающий на вход параметр IAsyncResult, передаваемый целевому методу делегата AsyncCallback, является экземпляром класса AsyncResult (заметьте, префикс I здесь отсутствует), определенного в пространстве имен System.Runtime. Remoting.Messaging. Статическое свойство AsyncDelegate возвращает ссылку на оригинальный асинхронный делегат, созданный где-то в программе. Таким образом, чтобы получить ссылку на объект делегата BinaryOp, размещенный в Main(), нужно просто преобразовать возвращенный свойством AsyncDelegate тип System.Object в тип BinaryOp. После этого можно вызвать EndInvoke(), как и ожидается.

**// Не забудьте добавить директиву 'using' для**

**// System.Runtime.Remoting.Messaging!**

static void AddComplete(IAsyncResult iftAR) {

 Console.WriteLine("Вызван AddComplete() в потоке {0}.", Thread.CurrentThread.GetHashCode());

 Console.WriteLine("Ваше сложение выполнено");

**// Теперь получим результат.**

 AsyncResult ar = (AsyncResult)itfAR;

 BinaryOp b = (BinaryOp)ar.AsyncDelegate;

 Console.WriteLine("10 + 10 равно {0}.",

 b.EndInvoke(itfAR));

}

**Передача и получение пользовательских данных состояния**

Заключительным аспектом нашего рассмотрения асинхронных делегатов будет обсуждение последнего из аргументов метода BeginInvoke() (этот аргумент у нас до сих пор был равен null). С помощью этого параметра можно передать в метод обратного вызова дополнительную информацию состояния из первичного потока. Ввиду того, что прототипом этого аргумента является System.Object, с его помощью можно передать практически любые данные, приемлемые для метода обратного вызова. Предположим для примера, что первичный поток должен передать методу AddComplete() пользовательское текстовое сообщение.

static void Main(string[] args) {

 …

 IAsyncResult iftAR = b.BeginInvoke(10, 10, new AsyncCallback(AddComplete), **"Main() благодарит вас за сложение этих чисел."**);

 …

}

Чтобы получить эти данные в контексте AddComplete(), используйте свойство AsyncState поступающего на вход параметра IAsyncResult.

static void AddComplete(IAsyncResult iftAR) {

 …

**// Получение объекта с информацией и преобразование его в строку.**

 string msg = (string)itfAR.AsyncState;

 Console.WriteLine(msg);

}

На рис. 14.4 показан вывод этого приложения.

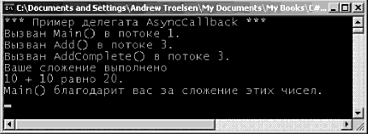


Рис. 14.4. Передача и получение пользовательских данных состояния

Чудесно! Теперь, когда вы понимаете, что делегат .NET можно использовать для автоматического запуска вторичного потока выполнения, обрабатывающего асинхронный вызов метода, давайте обратим внимание на возможности непосредственного взаимодействия о потоками с помощью пространства имен System.Threading.

**Исходный код**. Проект AsyncCallbackDelegate размещен в подкаталоге, соответствующем главе 14.

**Пространство имен System.Threading**

В рамках платформы .NET пространство имен System.Threading предлагает ряд типов, позволяющих строить многопоточные приложения. Вдобавок к типам, с помощью которых можно взаимодействовать с отдельными потоками CLR, в этом пространстве имен определены также типы, обеспечивающие доступ к поддерживаемому средой CLR пулу потоков, простой (не имеющий графического интерфейса) класс Timer и множество типов, предназначенных для поддержки синхронизированного доступа к разделяемым ресурсам. Описания основных членов этого пространства имен приведены табл. 14.1. (Не забывайте о том, что подробности всегда можно найти в документации .NET Framework 2.0 SDK.)

**Таблица 14.1.**Подборка типов пространства имен System.Threading

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| Interlocked | Предлагает атомарные операции для типов, открытых для множества потоков. |
| Monitor | Обеспечивает синхронизацию объектов потоков с помощью блокировок и ожиданий/сигналов, ключевое слово C# lock использует тип Monitor в фоновом режиме |
| Mutex | Примитив синхронизации, используемый для синхронизации взаимодействия между границами доменов приложения |
| ParameterizedThreadStart | Делегат (появившийся только в .NET 2.0), позволяющий потоку вызывать методы с любым числом аргументов |
| Semaphore | Позволяет ограничить число потоков, которые могут иметь конкурентный доступ к ресурсу или определенному типу ресурсов |
| Thread | Представляет поток, выполняющийся в среде CLR. С помощью этого типа можно создавать дополнительные потоки в оригинальном домене приложения |
| ThreadPool | Позволяет взаимодействовать о пулам потоков, управляемым средой CLR в рамках данного процесса |
| ThreadPriority | Перечень, представляющий уровень приоритета потока (Highest, Normal и т.д.) |
| ThreadStart | Делегат, используемый для указания метода, вызываемого для данного потока. В отличие от ParameterizedThreadStart, целевые методы ThreadStart должны соответствовать фиксированному шаблону |
| ThreadState | Перечень, указывающий состояния, допустимые для данного потока (Running, Aborted и т.д.) |
| Timer | Обеспечивает механизм выполнения метода через заданные интервалы времени |
| TimerCallback | Делегат, используемый в совокупности с типами Timer |

**Класс System.Threading.Thread**

Основным в пространстве имен System.Threading является класс Thread. Этот класс представляет собой объектный контейнер отдельной ветви выполнения в конкретном домене приложения. Он определяет ряд методов (как статических, так и общедоступных), которые позволяют создавать новые потоки в текущем домене приложения, а также приостанавливать, останавливать и завершать отдельные потоки. Рассмотрите описания основных статических членов, приведенные в табл. 14.2.

**Таблица 14.2.**Основные статические члены типа Thread

|  |  |
| --- | --- |
| **Статический член** | **Описание** |
| CurrentContext | Доступное только для чтения свойство, возвращающее контекст, в котором выполняется поток в настоящий момент |
| CurrentThread | Доступное только для чтения свойство, возвращающее ссылку на выполняемый в настоящий момент поток |
| GetDomain() GetDomainID() | Методы, возвращающие ссылки на текущий домен приложения или идентификатор домена, в котором выполняется текущий поток |
| Sleep() | Метод, приостанавливающий выполнение текущего потока на указанное время |

Класс Thread также поддерживает набор членов уровня экземпляра. Описания некоторых из этих членов приведены в табл. 14.3.

**Таблица 14.3.**Члены уровня экземпляра типа Thread

|  |  |
| --- | --- |
| **Член уровня экземпляра** | **Описание** |
| IsAlive | Возвращает логическое значение, сообщающее о том, запущен ли данный поток |
| IsBackground | Читает или устанавливает значение, сообщающее о том, является ли данный поток "фоновым" (дополнительные подробности будут предложены чуть позже) |
| Name | Позволяет задать понятное строковое имя потока |
| Priority | Читает или устанавливает приоритет потока, которому может быть назначено значение из перечня ThreadPriority |
| ThreadState | Читает информацию о состоянии потока, которая может принимать значения из перечня ThreadState |
| Abort() | Дает указание среде CLR завершить поток как можно быстрее |
| Interrupt() | Выводит (например, путем активизации) текущий поток из периода ожидания |
| Join() | Блокирует вызывающий поток до завершения указанного потока (того, для которого вызывается Join()) |
| Resume() | Возобновляет выполнение приостановленного ранее потока |
| Start() | Дает указание среде CLR как можно быстрее начать выполнение потока |
| Suspend() | Приостанавливает выполнение потока. Если поток уже приостановлен, вызов Suspend() игнорируется |

**Получение информации об отдельном потоке**

Напомним, что точка входа компоновочного блока (т.е. метод Main()) при выполнении оказывается в первичном потоке. Чтобы привести типичный пример использования типа Thread, предположим, что у нас есть новое консольное приложение с именем ThreadState. Вы знаете, что статическое свойство Thread.СurrentThread позволяет получить тип Thread, представляющий выполняемый в настоящий момент поток. Получив текущий поток, вы можете вывести на экран различную информацию о потоке.

**// Не забудьте указать 'using' для пространства имен System.Threading.**

static void Main(string[] args) {

 Console.WriteLine("\*\**\*\*\**Информация первичного потока \*\*\*\*\*\n");

**// Получение текущего потока и назначение ему имени.**

 Thread primaryThread = Thread.CurrentThread;

 primaryThread.Name = "ThePrimaryThread";

**// Подробности хостинга домена приложения и контекста.**

 Console.WriteLine("Имя текущего домена приложения: {0}";

 Thread.GetDomain().FriendlyName);

 Console.WriteLine("Идентификатор текущего контекста: {0}", Thread.CurrentContext.ContextID);

**// Вывод информации о данном потоке.**

 Console.WriteLine("Имя потока: {0}", primaryThreаd.Name);

 Console.WriteLine("Запущен ли поток? {0}", primaryThread.IsAlive);

 Console.WriteLine("Уровень приоритета: {0}", primaryThread.Priority);

 Console.WriteLine("Состояние потока: {0}", primaryThread.ThreadState);

 Console.ReadLine();

}

На рис. 14.5 показан вывод этого приложения.

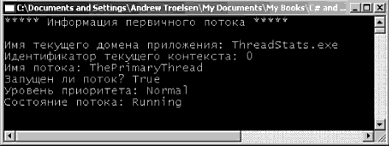


Рис. 14.5. Сбор статистики о потоке

**Свойство Name**

Приведенный выше программный код достаточно понятен, но обратите внимание на то, что класс Thread предлагает свойство с именем Name (имя). Если вы не установите для него значения, свойство Name будет возвращать пустую строку. Но, назначив данному объекту Thread в качестве имени понятную строку, вы можете сильно упростить процесс отладки. В Visual Studio 2005 в режиме отладки можно использовать окно Threads (Потоки), доступ к которому можно получить, выбрав Debug→Windows→Threads из меню. Как показано на рис. 14.6, в этом окне можно по имени идентифицировать поток, который следует проанализировать.

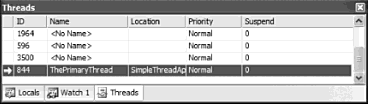


Рис. 14.6. Отладка потока в Visual Studio 2005

**Свойство Priority**

Далее заметим, что тип Thread определяет свойство с именем Priority. По умолчанию все потоки получают приоритет Normal (средний). Но вы можете изменить это значение в любой момент времени существования потока, используя свойство Priority и связанный с ним перечень System.Threading.ThreadPriority.

public enum ThreadPriority {

 AboveNormal,

 BelowNormal,

 Highest,

 Idle,

 Lowest,

**Normal, // Значение, используемое по умолчанию.**

 TimeCritical

}

При назначении потоку приоритета, отличного от принимаемого по умолчанию (ThreadPriority.Normal), вы должны понимать, что не обладаете слишком большими возможностями контроля в отношении того, когда планировщик потоков переключится с одного потока на другой. Уровень приоритета потока является лишь "подсказкой" среде CLR в отношении того, насколько важно выполнение данного потока. Поэтому поток со значением ThreadPriority.Highest (наивысший) не обязательно гарантирует данному потоку абсолютное преимущество.

Снова подчеркнем, что в том случае, когда планировщик потоков полностью занят текущей задачей (например, синхронизацией объекта, переключением или перемещением потоков), уровень приоритета будет, вероятнее всего, соответствующим образом изменен. Однако в других случаях соответствующие значения прочитает среда CLR, которая и выдаст планировщику потоков указания о том, как лучше всего организовать квантование времени. При прочих равных условиях потоки с идентичным приоритетом должны получать примерно одинаковое время для выполнения своей работы.

Необходимость изменения приоритетов потоков вручную возникает очень редко. Теоретически можно повысить приоритет для множества потоков так, что это не позволит потокам с более низкими приоритетами выполнять работу на их уровнях (поэтому используйте указанные возможности с осторожностью).

**Исходный код.**Проект ThreadState размещен в подкаталоге, соответствующем главе 14.

**Программное создание вторичных потоков**

Чтобы программно создавать дополнительные потоки, выполняющие свои отдельные задачи, вы должны следовать вполне понятным указанным ниже рекомендациям.

1. Для выбранного типа создайте метод, который будет использоваться в качестве точки входа нового потока.

2. Создайте делегат ParameterizedThreadStart (или уже устаревший ThreadStart), передав его конструктору адрес метода, определенного на шаге 1.

3. Создайте объект Thread, передав конструктору делегат ParameterizedThreadStart/ThreadStart в виде аргумента.

4. Задайте подходящие начальные характеристики потока (имя, приоритет и т.д.).

5. Вызовите метод Thread.Start(). Это указание как можно быстрее стартовать поток для метода, на который ссылается делегат, созданный на шаге 2.

Согласно шагу 2, имеется возможность использовать один из двух разных типов делегата для метода, предназначенного для выполнения во вторичном потоке. Делегат ThreadStart является частью пространства имен System.Threading со времен .NET версии 1.0 и может указывать на любой метод, не имеющий аргументов и не возвращающий ничего. Этот делегат удобно использовать тогда, когда метод должен выполняться в фоновом режиме без взаимодействия с ним.

Очевидным ограничением ThreadStart является отсутствие параметров. Поэтому в .NET 2.0 предлагается тип делегата ParameterizedThreadStart, допускающий передачу одного параметра типа System.Object. Поскольку с помощью System.Object можно представить всё, что угодно, вы можете передать этому делегату любое число параметров в виде пользовательского класса или структуры. Заметьте, однако, что делегат ParameterizedThreadStart может указывать только на методы, возвращающие void.

**Работа с делегатом ThreadStart**

Чтобы рассмотреть процесс создания многопоточного приложения на практику (а также продемонстрировать пользу соответствующего подхода), предположим, что у нас есть консольное приложение (SimpleMultiThreadApp), которое позволяет конечному пользователю выбрать в приложении либо использование одного первичного потока, выполняющего всю работу, либо разделение ее на два отдельных потока.

После того как вы обеспечите доступ к пространству имен System.Threading с помощью ключевого слова C# using, первым шагом должно быть определение метода, который будет выполнять работу во вторичном потоке. Чтобы сосредоточиться на сути механизма построения многопоточных программ, здесь этот метод просто выводит последовательность чисел с двухсекундными задержками перед каждой операцией вывода. Вот полное определение соответствующего класса Printer.

public class Printer {

 public void PrintNumbers() {

**// Отображение информации потока.**

  Console.WriteLine ("-› {0} выполняет PrintNumbers()", Thread.CurrentThread.Name);

**// Вывод чисел.**

  Console.Write("Ваши числа: ");

  for(int i = 0; i ‹ 10; i++) {

   Console.Write(i + ", ");

   Thread.Sleep(2000);

  }

  Console.WriteLine();

 }

}

Теперь в Main() нужно предложить выбор одного или двух потоков для выполнения задач приложения. Если пользователь выберет использование одного потока, просто вызывается метод PrintNumbers() в рамках первичного потока. Но если пользователь указывает два потока, создается делегат ThreadStart, указывающий на PrintNumbers(). Объект делегата передается конструктору нового объекта Thread и вызывается метод Start(), информирующий среду CLR о том, что поток готов к обработке.

Сначала установите ссылку на компоновочный блок System.Windows.Forms.dll и с помощью MessageBox.Show() отобразите подходящее сообщение в Main() (смысл этого станет ясным при запуске программы). Вот полная реализация Main() в нужном виде.

static void Main(string[] args) {

 Console.WriteLine("*\*\*\*\*\**Чудесное приложение Thread \*\*\*\*\*\n");

 Console.Write("Хотите иметь [1] или [2] потока?");

 string threadCount = Console.ReadLine();

**// Имя текущего потока.**

 Thread primaryThread = Thread.CurrentThread;

 primaryThread.Name = "Первичный";

**// Вывод информации Thread.**

 Console.WriteLine("-› {0} выполняет Main()", Thread.CurrentThread.Name);

***//*Создание рабочего класса.**

 Printer р = new Printer();

 switch (threadCount) {

 case "2":

**// Теперь создание потока.**

  Thread backgroundThread = new Thread(new ThreadStart(p.PrintNumbers));

  backgroundThread.Name = "Вторичный";

  backgroundThread.Start();

  break;

 case "1":

  p.PrintNumbers();

  break;

default:

 Console.WriteLine("Ваши указания не ясны… будет 1 поток.");

 goto case "1";

}

**// Выполнение дополнительней работы.**

MessageBox.Show("Я занят!", "Работа в главном потоке…");

Console.RеаdLine();

}

Если теперь запустить эту программу с одним потоком, вы обнаружите, что окно сообщения не будет отображено до тех пор, пока на консоль не будет выведена вся последовательность чисел. Здесь была указана пауза приблизительно в две секунды после вывода каждого из чисел, поэтому подобное поведение программы не вызовет восхищения конечного пользователя. Но если вы выберете вариант с двумя потоками, окно сообщения появится немедленно, поскольку для вывода чисел на консоль будет использоваться свой уникальный объект Thread (рис. 14.7).

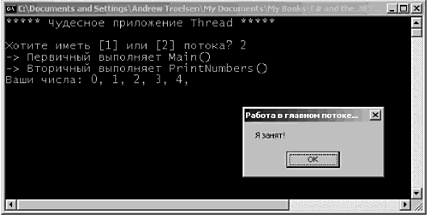


Рис. 14.7. Многопоточные приложения "более отзывчивы" при выдаче своих результатов

Здесь важно отметить, что при построении многопоточных приложений (с применением асинхронных делегатов) на машинах с одним процессором вы не получаете приложение, *выполняющееся*быстрее, чем позволяет процессор машины. При запуске этого приложения с использованием как одного, так и двух потоков числа будут отображаться одинаково. Многопоточные приложения позволяют улучшить*"отзывчивость"*приложения. Конечному пользователю может казаться, что такая программа работает быстрее, но на самом деле это не так. Потоки не имеют никакой возможности ускорить выполнение циклов foreach, операций вывода на печать или сложения чисел. Многопоточные приложения просто позволяют распределять нагрузку среди множества потоков.

**Исходный код.**Проект SimpleMultiThreadApp размещен в подкаталоге, соответствующем главе 14.

**Работа с делегатом ParameterizedThreadStart**

Напомним, что делегат ThreadStart может указывать только на методы, возвращающие void и не имеющие аргументов. Во многих случаях этого будет вполне достаточно, но передать данные методу, выполняющемуся во вторичном потоке, вы сможете только с помощью делегата ParameterizedThreadStart. Для примера воссоздадим программную логику проекта AsyncCallbackDelegate, построенного в этой главе выше, но на этот раз используем тип делегата ParameterizedThreadStart.

Сначала создайте новое консольное приложение AddWithThreads и укажите using для пространства имен System.Threading. Поскольку ParameterizedThreadStart может указывать на любой метод, принимающий параметр System.Object, создайте пользовательский тип, содержащий числа для сложения.

class AddParams {

 public int a;

 public int b;

 public AddParams(int numb1, int numb2) {

  a = numb1;

  b = numb2;

 }

}

В классе Program создайте статический метод, который с помощью типа AddParams напечатает сумму соответствующих значений.

public static void Add(object data) {

 if (data is AddParams) {

  Console.WriteLine("ID потока в Add(): {0}", Thread.CurrentThread.GetHashCode());

  AddParams ap = (AddParams)data;

  Console.WriteLine("{0} + {1} равно {2}", ар.a, ар.b, ар.a + ар.b);

 }

}

Программный код Main() в данном случае предельно прост. Просто используйте ParameterizedThreadStart вместо ThreadStart.

static void Main(string[] args) {

 …

 Console.WriteLine("\*\*\*\*\* Сложение с объектами Thread \*\*\*\*\*");

 Console.WriteLine("ID потока в Main(): {0}", Thread.CurrentThread.GetHashCode());

 AddParams ap = new AddParams(10, 10);

 Thread t = new Thread(new ParameterizedThreadStart(Add));

 t.Start(ap);

 …

}

Исходный код. Проект AddWithThreads размещен в подкаталоге, соответствующем главе 14.

**Приоритетные и фоновые потоки**

Итак, вы научились программного создавать новые потоки выполнения с помощью пространства имен System.Threading, теперь давайте выясним, чем отличаются приоритетные и фоновые потоки.

• *Приоритетные потоки*обеспечивают текущему приложению защиту от преждевременного завершения. Среда CLR не прекратит работу приложения (лучше сказать, не выгрузит соответствующий домен приложения), пока не завершат работу все приоритетные потоки,

• *Фоновые потоки*(иногда называемые *демонами)*рассматриваются средой CLR, как возобновляемые ветви выполнения, которыми можно пренебречь в любой момент времени (даже при выполнении ими своих задач). Поэтому, когда все приоритетные потоки завершаются, все фоновые потоки будут завершены автоматически в результате выгрузки домена приложения.

Важно понять, что понятия приоритетного и фонового потоков – это *не*синонимы понятий первичного и рабочего потока. По умолчанию каждый поток, создаваемый с помощью метода Thread.Start(), автоматически оказывается *приоритетным*потоком. А это значит, что домен приложения не будет выгружен до тех пор, пока в нем все потоки не завершат свою работу. В большинстве случаев это будет именно тем поведением, которое требуется.

Но предположим, что нам нужно вызвать Printer.PrintNumbers() во вторичном потоке, который должен действовать, как фоновый поток. Это означает, что для метода, на который указывает тип Thread (посредством делегата ThreadStart или ParameterizedThreadStart), должна допускаться возможность безболезненного его завершения, как только все приоритетные потоки закончат свою работу. Для настройки такого потока достаточно установить значение true (истина) для свойства IsBackground.

static void Main(string[] args) {

 Printer p = new Printer();

 Thread bgroundThread = new Thread(new ThreadStart(p.PrintNumbers));

  bgroundThread.**IsBackground**= true;

  bgroundThread.Start();

}

Обратите внимание на то, что метод Main() здесь не вызывает Console.ReadLine(), чтобы гарантировать присутствие консоли на экране до нажатия клавиши «Enter». Поэтому при выполнении этого приложения оно сразу же прекратит свою работу, так как объект Thread сконфигурирован для работы в фоновом потоке. С началом работы метода Main() создается *приоритетный*первичный поток, поэтому, как только выполнение программной логики Main() завершится, домен приложения будет выгружен, и это произойдет до того, как вторичный поток завершит свою работу. Однако, закомментировав строку, в которой устанавливается свойство IsBackground, вы обнаружите, что на консоль выводятся все числа, поскольку для того, чтобы домен приложения будет выгружен из содержащего его процесса, все приоритетные потоки должны завершить свою работу.

Обычно конфигурация потока для выполнения в фоновом режиме может быть полезна тогда, когда соответствующий рабочий поток выполняет некритичные задания, которые оказываются не нужными после завершения выполнения главной задачи программы.

**Исходный код.**Проект BackgroundThread размещен в подкаталоге, соответствующем главе 14.

**Проблема конкурентного доступа**

До сих пор все многопоточные приложения, созданные вами при изучении материала этой главы, были устойчивыми в отношении потоков, поскольку в них соответствующие методы вызывались только одним объектом Thread. Конечно, некоторые из ваших приложений могут быть настолько же простыми, но большинство многопоточных приложений содержит очень много вторичных потоков. С учетом того, что все потоки в домене приложения могут претендовать на доступ к открытым данным приложения одновременно, представьте себе, что может случиться, если к одному и тому же элементу данных получит доступ множество потоков. Поскольку планировщик потоков может приостановить работу потока в любой момент времени, что будет, если поток А будет отстранен от выполнения своей работы на полпути до того, как он эту работу завершит? Поток В будет читать некорректные данные.

Чтобы проиллюстрировать проблему конкурентного доступа, давайте построим еще одно консольное приложение C#, которое мы назовем MultiThreadedPrinting, Это приложение будет использовать класс Printer, созданный нами ранее, но на этот раз метод PrintNumbers() "заставит" текущий поток делать паузы произвольной длительности в соответствии со случайно генерируемыми значениями.

public class Printer {

 public void PrintNumbers() {

  …

  for (int i = 0; i ‹ 10; i++) {

   Random r = new Random();

   Thread.Sleep(1000 \* r.Next(5));

   Console.Write(i + ", ");

  }

  Console.WriteLine();

 }

}

Метод Main() отвечает за создание массива из десяти объектов Thread с уникальными именами), каждый из который вызывает один и тот же экземпляр Printer.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Синхронизация потоков \*\*\*\*\*\n");

  Printer p = new Printer();

**// Создание 10 потоков, указывающих на один и тот же метод**

**// одного и того же объекта.**

 Thread[] threads = new Thread[10];

  for (int i = 0; i ‹ 10; i++) {

   threads[i] =new Thread(new ThreadStart(p.PrintNumbers));

   threads[i].Name = string.Format("Рабочий поток #{0}", i);

  }

**// Теперь старт каждого их них.**

  foreach (Thread t in threads) t.Start();

  Console.ReadLine();

 }

}

Перед тем как выполнить тестовый запуск программы, давайте обсудим cо-ответствующую проблему. Здесь первичный поток в рамках домена приложения порождает десять вторичных рабочих потоков. Каждому рабочему потоку дается указание вызвать метод PrintNumbers() одного и того же экземпляра Printer. Поскольку здесь не предпринято никаких мер по блокированию общедоступных ресурсов данного объекта (консоли), имеется большая вероятность того, что текущий поток будет приостановлен до того, как метод PrintNumbers() закончит вывод всех своих результатов. Вы не знаете точно, когда это случиться (и случится ли вообще), поэтому нужно быть готовым к непредвиденным результатам. Например, может получиться вывод, показанный на рис. 14.8.

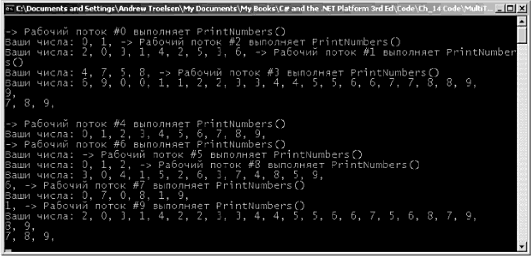


Рис. 14.8. Конкуренция в действии, первая попытка

Выполните приложение еще несколько раз. На рис. 14.9 показана другая возможность вывода (ваши результаты, очевидно, тоже будут другими).

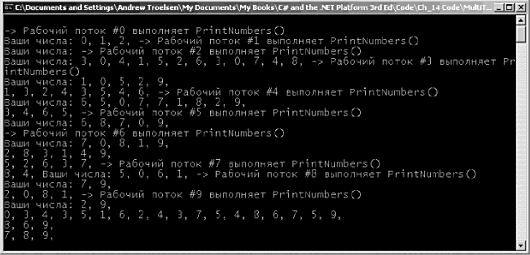


Рис. 14.9. Конкуренция в действии, вторая попытка

Ясно, что проблемы здесь действительно есть. Каждый поток дает указание объекту Printer печатать числовые данные, и планировщик потоков запускает выполнение этих потоков в фоновом режиме. В результате получается несогласованный вывод. В этом случае мы должны программно организовать синхронизованный доступ к совместно используемым ресурсам. Нетрудно догадаться, что в пространстве имен System.Threading есть целый ряд типов, имеющих отношение к синхронизации. А язык программирования C# предлагает специальное ключевое слово, как раз для решения задач синхронизации совместного доступа к данным в многопоточных приложениях.

**Замечание**. Если у вас не получается сгенерировать непредвиденный вывод, увеличьте число потоков с 10 до 100 (например) или добавьте в свою программу вызов Thread.Sleep(). В конце концов вы все равно столкнетесь с проблемой конкурентного доступа

**Синхронизация с помощью ключевого слова lock в C#**

Первой из возможностей, которую вы можете применить в C# для синхронизации доступа к совместно используемым ресурсам, является использование ключевого слова lock. Это ключевое слово позволяет определить контекст операторов, которые должны синхронизироваться между потоками. В результате входящие потоки не смогут прервать текущий поток, пока он выполняет свою работу. Ключевое слово lock требует, чтобы вы указали *маркер*(объектную ссылку), который потребуется потоку для входа в пределы контекста lock. При блокировке метода уровня экземпляра можно использовать просто ссылку на текущий тип.

**// Использование текущего объекта в качестве маркера потока.**

lock(this) {

**// Весь программный код в этом контексте оказывается**

**// устойчивым в отношении потоков.**

}

При внимательном изучении метода PrintNumbers() становится ясно, что совместно используемым ресурсом, за доступ к которому соперничают потоки, является окно консоли. Поместите в рамки соответствующего контекста блокировки все операторы взаимодействии с типом Console так, как показано ниже.

public void PrintNumbers() {

 lock (this) {

**// Вывод информации Thread.**

  Console.WriteLine("-› {0} выполняет PrintNumbers()", Thread.CurrentThread.Name);

**// Вывод чисел.**

  Console.Write("Ваши числа": ");

  for (int i = 0; i ‹ 10; i++) {

   Random r *=*new Random();

   Thread.Sleep(1000 \* r.Next(5));

   Console.Write(i + ", ");

  }

  Console.WriteLine();

 }

}

Тем самым вы создадите метод, который позволит текущему потоку завершить выполнение своей задачи. Как только поток вступит в контекст блокировки, соответствующий маркер блокировки (в данном случае эта ссылка на текущий объект) станет недоступным другим потокам, пока блокировка не будет снята в результате выхода потока из контекста блокировки. Например, если маркер блокировки получает поток А, то другие потоки не смогут войти в контекст до тех пор, пока поток А не освободит маркер блокировки.

**Замечание.**Если пытаться блокировать программный код в статическом методе, вы, очевидно, не можете использовать ключевое слово this. Но в этом случае можно передать объект System.Type соответствующего класса с помощью оператора C# typeof.

Если снова выполнить это приложение, вы увидите, что теперь каждый поток получает возможность закончить свою работу (рис. 14.10).

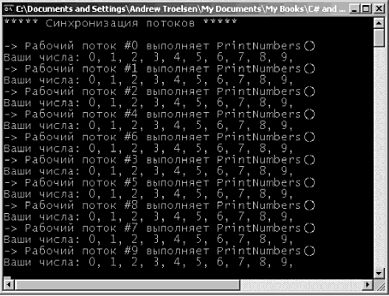


Рис. 14.10. Конкуренция в действии, третья попытка

**Исходный код.**Проект MultiThreadedPrinting размещен в подкаталоге, соответствующем главе 14.

**Синхронизация с помощью типа System.Threading.Monitor**

Оператор C# lock на самом деле является лишь ключевым словом, обозначающим использование типа класса System.Threading.Monitor. После обработки компилятором C# контекст блокировки превращается в следующее (вы можете убедиться в этом с помощью ildasm.exe).

public void PrintNumbers() {

**Monitor.Enter(this)**;

 try {

**// Вызов информации Thread.**

  Console.WriteLine("-› {0} выполняет PrintNumbers()", Thread.CurrentThread.Name);**// Вывод чисел.**

  Console.Write("Ваши числа: ");

  for (int i = 0; i ‹ 10; i++) {

   Random r = new Random();

   Thread.Sleep(1000\* r.Next(5));

   Console.Write(i + ", ");

  }

  Console.WriteLine();

 } finallу {

**Monitor.Exit(this);**

 }

}

Во-первых, заметим, что конечным получателем маркера потока, который был указан в качестве аргумента ключевого слова lock, является метод Monitor.Enter(). Во-вторых, весь программный код в рамках контекста соответствующей блокировки помещен в блок try. Соответствующий блок finally гарантирует, что маркер потока будет освобожден (с помощью метода Monitor.Exit()), независимо от исключений, которые могут возникать в среде выполнения. Если изменить программу MultiThreadSharedData так, чтобы тип Monitor использовался непосредственно (как это будет сделано чуть позже), то ее вывод останется тем же.

При использовании ключевого слова lock, кажется, требуется меньший ввод программного кода, чем при явном использований типа System.Threading.Monitor, поэтому вы можете задать вопрос о преимуществах непосредственного использования типа Monitor. Краткий ответ: контроль. При использовании типа Monitor вы можете дать указание активному потоку подождать (с помощью метода Wait()), информировать ожидающие потоки о завершении текущего потока (с помощью методов Pulse() и PulseAll()) и т.д.

В большинстве случаев вам будет вполне достаточно возможностей, обеспечиваемых ключевым словам C# lock. Но если вы захотите рассмотреть другие члены класса Monitor, обратитесь к документации .NET Framework 2.0 SDK.

**Синхронизация с помощью типа System.Threading.Interlocked**

В это всегда верится с трудом, пока вы не проверите соответствующий программный код CIL, но и операции присваивания, и базовые арифметические операции *не являются атомарными.*Поэтому в пространстве имен System.Threading предлагается тип, позволяющий воздействовать на отдельный элемент данных атомарно с меньшей нагрузкой, чем это делает тип Monitor. Тип класса Interlocked определяет статические члены, описания которых приведены в табл. 14.4.

**Таблица 14.4.**Члены типа System.Threading.Interlocked

|  |  |
| --- | --- |
| **Член** | **Описание** |
| CompareExchange() | Безопасно проверяет два значения на равенство, и если они равны, заменяет одно из значений третьим |
| Decrement() | Безопасно уменьшает значение на 1 |
| Exchange() | Безопасно меняет два значения местами |
| Increment() | Безопасно выполняет приращение значения на 1 |

Хотя это может и не казаться очевидным на первый взгляд, процесс атомарного изменения одного значения является вполне типичным в многопоточном окружении. Предположим, что у нас есть метод AddOne(), который увеличивает целочисленную переменную intVal на единицу. Вместо программного кода синхронизации, подобного следующему;

public void AddOne() {

 lock(this) {

  intVal++;

 }

}

можно предложить более простой программный код, в котором используется статический метод Interlocked.Increment(). Просто передайте переменную для приращения по ссылке. Обратите внимание на то, что метод Increment() не только изменяет значение поступающего параметра, но и возвращает новое значение.

public void AddOne() {

 int newVal = **Interlocked.Increment**(ref intVal);

}

В дополнение к Increment() и Decrement() тип Interlocked позволяет атомарно присваивать числовые и объектные данные. Например, если вы хотите присвоить члену-переменной значение 83, вы можете избежать необходимости явного использования оператора lock (или явного применения логики Monitor), если используете метод Interlocked.Exchange().

public void SafeAssignment() {

**Interlocked.Exchange**(ref myInt, 83);

}

Наконец, при проверке двух значений на равенство, чтобы обеспечить потоковую безопасность элементу сравнения, можете использовать метод Interlocked.CompareExchange(), как показано ниже.

public void CompareAndExchange() {

**// Если значением i является 83, изменить его на 99.**

 Interlocked.**CompareExchange**(ref i, 99, 83);

}

**Синхронизация с помощью атрибута [Synchronization]**

Последним из рассмотренных здесь примитивов синхронизации будет атрибут [Synchronization], который определяется в пространстве имен System.Runtime.Remoting.Contexts. Этот атрибут уровня класса для безопасности потока эффективно блокирует *весь*программный код членов экземпляра. Когда среда CLR размещает объект, имеющий атрибут [Synchronization], она помещает этот объект в рамки синхронизированного контекста. Вы должны помнить из главы 13, что объекты, которые не должны покидать контекстные границы, являются производными от ContextBoundObject. Поэтому, чтобы сделать тип класса Printer устойчивым в отношении потоков (без добавления программного кода защиты потоков вручную), следует изменить соответствующее определение так.

using System.Runtime.Remoting.Contexts;

...

**// Все методы Printer теперь потокоустойчивы!**

[Synchronization]

public class Printer: СontextBoundObject {

 public void PrintNumbers() {

 …

 }

}

Этот подход можно назвать способом создания потокоустойчивого программного кода для ленивых, поскольку здесь не требуется выяснять, какие фрагменты типа могут испытывать влияние внешних потоков. Главным недостатком этого подхода является то, что даже если какой-то метод и не испытывает влияния внешних потоков, среда CLR*все равно*блокирует обращение к этому методу. Очевидно, это может ухудшить общие характеристики функционирования типа, так что используйте указанный подход с осторожностью.

Итак, мы с вами обсудили целый ряд подходов к решению вопроса синхронизированного доступа к общим блокам данных. Будьте уверены, в пространстве имен System.Threading есть и другие типы, которые я настоятельно рекомендую вам постепенно исследовать. В завершение этой главы, посвященной программированию потоков, давайте рассмотрим три: дополнительных типа: TimerCallback, Timer и ThreadPool.

**Программирование с помощью таймеров обратного вызова**

Во многих приложениях возникает необходимость вызывать конкретный метод через регулярные промежутки времени. Например, в одном приложении может потребоваться отображение текущего времени в строке состояния с помощью некоторой вспомогательной функции. В другом приложении может понадобиться периодический вызов вспомогательной функции, выполняющей в фоновом режиме какие-то некритические задачи, например проверку поступления новых сообщений электронной почты. Для таких ситуаций можно использовать тип System. Threading.Timer в совокупности с соответствующим делегатом TimerCallback.

Для примера предположим, что нам нужно создать консольное приложение, которое ежесекундно выводит текущее время, пока пользователь не нажмет клавишу, завершающую выполнение этого приложения. Первым очевидным шагом здесь является создание метода, который будет вызываться типом Timer.

class TimePrinter {

 static void PrintTime(object state) {

  Console.WriteLine("Время: {0}", DateTime.Now.ToLongTimeString());

 }

}

Этот метод имеет один параметр типа System.Object и возвращает void. Такая структура метода обязательна, поскольку делегат TimerCallback может вызывать только такие методы. Значение, передаваемое целевому методу делегата TimerCallback, может представлять любую информацию (так, в случае электронной почты это может быть имя сервера Microsoft Exchange, с которым требуется взаимодействие в ходе процесса). А так как параметр является типом System.Object, в действительности можно передать любое число аргументов, если использовать System.Array или пользовательский класс (структуру).

Следующим шагом является настройка экземпляра делегата TimerCallback и передача его объекту Timer. Кроме делегата TimerCallback, конструктор Timer позволяет указать дополнительную информацию (в виде System.Object) для передачи ее целевому объекту делегата, временной интервал опроса метода и время ожидания (в миллисекундах) до начала первого вызова, например:

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Работа с типом Timer \*\*\*\*\*\n");

**// Создание делегата для типа Timer.**

 TimerCallback timeCB = new TimerCallback(PrintTime);

**// Установка параметров таймера.**

 Timer t = new Timer(

  timeCB, // **Тип делегата TimerCallback.**

  null, // **Информация для вызываемого метода или null.**

  0, **// Время ожидания до старта.**

  1000); // **Интервал между вызовами (в миллисекундах)**.

 Console.WriteLine("Нажмите «Enter» для завершения работы…");

 Console.ReadLine();

}

В данном случае метод PrintTime() будет вызываться примерно каждую секунду и методу не передается никакой дополнительной информации. Чтобы передать целевому объекту делегата какую-то информацию, замените значение null второго параметра конструктора подходящим значением (например, "Привет"). Следующая модификация метода PrintTime() использует переданное значение.

static void PrintTime(Object state) {

 Console.WriteLine("Время: {0}, Параметр: {1}", DateTime.Now.ToLongTimeString(), state.ToString());

}

На рис. 14.11 показан соответствующий вывод.

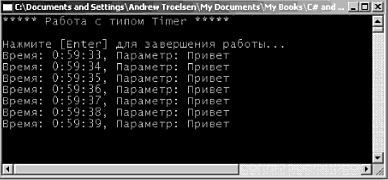


Рис. 14.11. Таймеры за работой

**Исходный код**. Проект TimerApp размещен в подкаталоге, соответствующем главе 14.

**Пул потоков CLR**

Заключительной темой нашего обсуждения в этой плаве, посвященной потокам, будет пул потоков CLR. При асинхронном вызове типов с помощью делегатов (посредством метода BeginInvoke()) нельзя сказать, что среда CLR буквально создает совершенно новый поток. В целях эффективности метод BeginInvoke() делегата использует пул (динамическую область) рабочих потоков, поддерживаемых средой выполнения. Чтобы позволить вам взаимодействовать с этим пулом рабочих потоков, пространство имен System.Threading предлагает тип класса ThreadPool.

Чтобы поставить вызов метода в очередь для обработки рабочим потоком из пула, используйте метод ThreadPool.QueueUserWorkItem(). Этот метод является перегруженным, чтобы вдобавок к экземпляру делегата WaitCallback имелась возможность указать необязательный System.Objеct для пользовательских данных состояния.

public sealed class ThreadPool {

 …

 public static bool QueueUserWorkItem(WaitCallback callBack);

 public static bool QueueUserWorkItem(WaitCallback callBack, **object state**);

}

Делегат WaitCallback может указывать на любой метод, имеющий один параметр System.Object (для представления необязательных данных состояния) и не возвращающий ничего. Если при вызове QueueUserWorkItem() вы не предложите System.Object, среда CLR автоматически передаст значение null. Для иллюстрации методов очереди при использовании пула потоков CLR давайте рассмотрим следующую программу, в которой снова используется тип Printer. Но на этот раз мы не будем создавать массив типов Thread вручную, а свяжем метод PrintNumbers() с членами пула.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("Старт главного потока. ThreadID = {0}", Thread.CurrentThread.GetHashCode());

  Printer p = new Printer();

  WaitCallback workItem = new WaitCallback(PrintTheNumbers);

**// Очередь из 10 вызовов метода.**

  for (int i = 0; i ‹ 10; i++) {

   ThreadPool.QueueUserWorkItem(workItem, p);

  }

  Console.WriteLine("Все задачи в очереди");

  Console.ReadLine();

 }

 static void PrintTheNumbers(object state) {

  Printer task = (Printer)state;

  task.PrintNumbers();

 }

}

Здесь вы можете спросить, разве выгодно использовать поддерживаемый средой CLR пул потоков вместо явного создания объектов Thread? Тогда рассмотрите следующие главные преимущества использования пула.

• Пул потоков управляет потоками эффективнее, поскольку минимизируется число потоков, которые приходится создавать, запускать и останавливать.

• При использовании пула потоков вы можете сосредоточиться на своей конкретной задаче, не отвлекаясь на вопросы инфраструктуры потоков приложения.

Однако управление потоками "вручную" может оказаться предпочтительнее, например, в следующих случаях.

• Если требуется создавать приоритетные потоки или устанавливать приоритеты потоков. Потоки, помещенные в пул, всегда являются фоновыми потоками с обычным уровнем приоритета (ThreadPriority.Normal).

• Если требуется создать поток с фиксированным идентификатором, чтобы име-лаcь возможность завершить, приостановить или обнаружить его по имени.

**Исходный код**. Проект ThreadPoolApp размещён в подкаталоге, соответствующем главе 14.

На этом наш экскурс в многопоточное программирование .NET завершается. Пространство имен System.Threading, без сомнения, определяет множество других типов, кроме тех*,*которые смогли уместиться в рамках обсуждения данной главы. Но сейчас вы имеете прочный фундамент, который позволит вам расширять свой знания.

**Резюме**

Эта глава началась с рассмотрения того, как настроить тип делегата .NET на вызов методов в асинхронной форме. Как было показано, методы BeginInvoke() и EndInvoke() позволяют косвенно управлять фоновыми потоками с минимальными усилиями и практически без проблем. В ходе обсуждения были рассмотрены интерфейс IAsyncResult и тип класса AsyncResult. Эти типы обеспечивают различные способы синхронизации вызовов и получения возвращаемых значений методов.

Оставшаяся часть главы была посвящена выяснению роли пространства имен System.Threading. Вы узнали о том, что в результате создания приложением дополнительных потоков программа получает (мнимую) возможность выполнять множество задании одновременно. Были рассмотрены различные способы защиты блоков программного кода, уязвимых в отношении потоков, чтобы при совместном использовании ресурсов потоками не происходило повреждения данных. Наконец, вы узнали о том, что среда CLR поддерживает пул потоков с целью повышения общей производительности системы и удобства ее использования.

**ГЛАВА 15. CIL и роль динамических компоновочных блоков**

В этой главе ставится две задачи. В первой половине главы будет рассмотрен синтаксис и семантика языка CIL (Common Intermediate Language – общий промежуточный язык) намного более подробно, чем в предыдущих главах. Честно говоря, при создании программ .NET вполне можно обойтись и без непосредственного изучения подробностей внутреннего устройства CIL-кода. Однако, изучив основы CIL, вы получите более глубокое понимание того, как функционируют некоторые "магические" особенности .NET (например, межъязыковое наследование). В оставшейся части главы будет исследована роль пространства имен System. Reflection.Emit. Используя его типы, вы получаете возможность строить программное обеспечение, способное генерировать компоновочные блоки .NET в памяти во время выполнения. Формально компоновочные блоки, определенные и выполняемые в памяти, называют *динамическими компоновочными блоками.*Как вы можете догадаться, эта специальная возможность .NET требует знания языка CIL, поскольку от вас потребуется указать набор CIL-инструкций, которые будут использоваться при создании компоновочного блока.

**Природа программирования в терминах CIL**

CIL – это родной язык платформы .NET, Когда вы создаете компоновочный блок .NET, используя тот управляемый язык, который вы предпочитаете, соответствующий компилятор переводит ваш исходный код в термины CIL. Подобно любому языку программирования, язык CIL предлагает множество программных и структурных лексем. Поскольку CIL является одним из языков программирования .NET не должно быть удивительным то, что вполне возможно создавать компоновочные блоки .NET непосредственно с помощью CIL и CIL-компилятора (ilasm.exe), вхо-дящего в стандартную поставку .NET Framework 2.0 SDK.

Хотя вполне очевидно, что лишь немногие программисты предпочтут строить свои .NET-приложения непосредственно на языке CIL, язык CIL сам по себе является чрезвычайно интересным объектом для интеллектуального исследования. Проще говоря, чем лучше вы понимаете грамматику CIL, тем увереннее вы будете себя чувствовать в мире нетривиальных приемов разработки .NET. Если говорить конкретно, то разработчик, обладающий пониманием языка CIL, получает следующее.

• Понимание того, как различные языки программирования .NET проецируют свои ключевые слова в лексемы CIL.

• Возможность дезассемблирования компоновочных блоков .NET, редактирования программного кода CIL и перекомпиляции обновленного базового кода в измененный двоичный код  .NET

• Возможность построения динамических компоновочных блоков с помощью элементов пространства имен System.Refleсtion.Emit.

• Иcпользование тех возможностей CTS (Common Type System – общая система типов), которые не поддерживаются управляемыми языками более высокого уровня, но существуют на уровне CIL. Язык CIL является единственным языком .NET, позволяющим получить доступ ко всем возможностям CTS.

Например, используя CIL, вы можете определять члены и поля глобального уровня (что не позволено в C#).

Снова заметим, чтобы было предельно ясно, что если вы *не хотите*углубляться в детали внутреннего устройства программного кода CIL, вам может быть вполне достаточно освоения возможностей библиотек базовых классов .NET. Во многих отношениях роль понимания языка CIL аналогична роли понимания языка ассемблера программистом, использующим C(++). Тем, кто понимает низкоуровневые возможности, проще находить хитроумные решения сложных задач с учетом тонких требований среды программирования (и среды выполнения). Так что если вы готовы принять вызов, давайте приступим к. рассмотрению особенностей CIL.

**Замечание**. Следует понимать, что в данной главе не предлагается всестороннее и исчерпывающее описание синтаксиса и семантики CIL. Если вам требуется всесторонний анализ возможностей CIL, обратитесь к книге Jason Bock, *CIL Programming: Under the Hood of .NET*(Apress, 2002).

**Директивы, атрибуты и коды операций CIL**

В начале изучения нового языка низкого уровня, такого как CIL, вы непременно обнаружите новые для себя (а часто и кажущиеся нелогичными) имена для очень привычных понятий. Рассмотрите, например, следующий набор элементов.

{new, public, this, base, get, set, explicit, unsafe, enum, operator, partial}

Вы, скорее всего, идентифицируете их, как ключевые слова языка C# (и это правильно). Но если присмотреться к элементам этого набора более внимательно, вы сможете заметить, что хотя здесь каждый элемент и является ключевым словом C#, они имеют совершенно разную семантику. Например, ключевое слово enum определяет тип, производный от System.Enum, а ключевые слова this и base позволяют ссылаться, соответственно, на текущий объект или родительский класс объекта. Ключевое слово unsafe используется для создания блока программного вода, который не должен непосредственно контролироваться средой CLR, а ключевое слово operator позволяет построить скрытый (специально именованный) метод, который будет вызываться тогда*,*когда вы применяете заданный оператор C# (например, знак сложения).

В отличие от такого высокоуровневого языка, как C#, язык CIL не просто определяет свой собственный набор ключевых слов. Набор лексем, понятных компилятору CIL, разделяется на три большие категории, в зависимости от семантического подтекста:

• директивы CIL;

• атрибуты CIL;

• коды операций CIL.

Каждая категория лексем CIL выражается с помощью своих специальных синтаксических конструкций, а сами лексемы объединяются с тем, чтобы в результате получился работоспособный компоновочный блок .NET.

**Роль директив CIL**

Прежде всего, есть множество известных лексем CIL, которые используются для описания полной структуры компоновочного блока .NET. Эти лексемы называются *директивами.*Директивы CIL используются дли информирования компилятора CIL о том, как определять пространства имен, типы и члены, содержащиеся в компоновочном блоке.

Синтаксически директивы обозначаются с помощью префикса, представленного точкой (.) (например, .namespace, .class, .publickeytoken, .override, .method, .assembly и т.д.). Так, если ваш файл \*.il (обычное расширение для файла, содержащего программный код CIL) имеет одну директиву .namespace и три директивы .сlass, компилятор CIL сгенерирует компоновочный блок, который определит одно пространства имен *.*NET и три типа класса .NET.

**Роль атрибутов CIL**

Во многих случаях директивы CIL сами по себе оказываются недостаточно информативными, чтобы дать исчерпывающее определение соответствующего типа .NET или его члена. Поэтому многие директивы CIL сопровождаются различными *атрибутами*CIL, сообщающими о том, как должна обрабатываться данная директива. Например, директива .class может сопровождаться атрибутам public (чтобы задать параметры видимости типа), атрибутом extends (чтобы явно указать базовый класс типа) или атрибутом implements (чтобы задать список интерфейсов, поддерживаемых типом).

**Роль кодов операций CIL**

После определения компоновочного блока .NET, пространства имен и набора типов в терминах GIL с использованием различных директив и связанных атрибутов остается одно – предложить программную логику реализации типа. Это является задачей *кодов операций.*В соответствии с традициями других языков низкого уровня, коды операций CIL, как правило, имеют просто непроизносимые аббревиатуры. Например, чтобы определить переменную строки, используется не понятный код операции LoadString, a ldstr.

Но все же, что не может не радовать, некоторые коды операций CIL в точности соответствуют их аналогам в C# (это, например, box, unbox, throw и sizeof). Вы сможете убедиться в том, что коды операций CIL всегда используются в контексте реализации члена и, в отличие от директив CIL, они никогда не обозначаются префиксом, заданным точкой.

**Различия между мнемоникой и кодом операции CIL**

Как только что объяснялось, коды операций, например ldstr, используются для реализации членов данного типа. Но в реальности лексемы (в том числе и ldstr) являются *мнемониками*CIL, представляющими на самом деле *двоичные коды операций*CIL. Чтобы пояснить различие, предположим, что у нас есть следующий метод, созданный средствами C#.

static int Add(int x, int у) {

 return х + у;

}

В терминах CIL сложение двух чисел представлено кодом операции 0X58. Аналогично для представления вычитания используется код операции 0X59, а действие, соответствующее размещению нового объекта в управляемой динамической памяти, обозначается кодом операции 0X73. С учетом сказанного должно быть ясно, что CIL-код, обрабатываемый JIT-компилятором, на самом деле является набором двоичных данных.

К счастью, для каждого двоичного кода операции CIL есть соответствующая мнемоника. Например, мнемоника add может использоваться вместо 0X58, sub – вместо 0X59, a newobj – вместо 0X73. Ввиду указанных различий между мнемониками и кодами операций, нетрудно догадаться, что декомпиляторы CIL, такие как, например, ildasm.exe, переводят двоичные коды операций компоновочного блока в соответствующую мнемонику CIL.

.method public hidebysig static int32 Add(int32 x, int32 y) cil managed {

 …

**// Лексема 'add' является более понятной мнемоникой CIL,**

**// используемой для представления кода операции 0X58.**

 add

 …

}

Тем, кто не сталкивается с необходимостью разработки низкоуровневого программного обеспечения .NET (например, пользовательского управляемого компилятора), обычно не приходится иметь дело непосредственно с числовыми кодами операций CIL. Поэтому практически всегда, когда программисты .NET говорят о "кодах операций CIL", они (как и я в этом тексте) имеют в виду набор более понятной мнемоники, а не лежащие в ее основе двоичные значения.

**Добавление и извлечение данных: стековая природа CIL**

Высокоуровневые языки .NET (например, такие как C#) пытаются максимально скрыть низкоуровневые сложности. Одним из аспектов разработки .NET, который оказывается скрытым особенно хорошо, является тот факт, что CIL является языком, целиком основанным на стековом программировании. Напомним, что при исследований пространства имен System.Collections (см. главу 7) мы с вами выяснили, что тип stack может использоваться для добавления значения в стек, а также для удаления из стека значения, размещенного на вершине стека. Конечно, разработчики CIL-приложений для загрузки и выгрузки значений не используют непосредственно объект System.Сollections.Stack, однако они применяют аналогичные операции.

Формально объект, используемый для хранения набора значений, называется *виртуальным стеком*выполнения. Вы сможете убедиться в том, что CIL предлагает целый ряд кодов операций, которые используются для добавления значения в стек: соответствующий процесс называется загрузкой. Точно так же CIL определяет целый ряд других кодов операций, которые переносят значение с вершины стека в память (например, в локальную переменную): для обозначения этого процесса используется термин *сохранение.*

В CIL просто невозможно получить доступ к элементам данных непосредственно, и это касается как локально определенных переменных, так и входных аргументов методов, а также полей данных типов. Нужно сначала явно загрузить элемент в стек, чтобы затем "вытолкнуть" его оттуда для дальнейшего использования (помните об этом, ведь именно поэтому блок программного кода CIL может казаться несколько избыточным).

Чтобы понять, как CIL использует стековую модель, рассмотрим простой C#-метод PrintMessage(), который не имеет аргументов и ничего не возвращает. В рамках реализации этого метода вы просто выводите значение локальной строковой переменной в поток стандартного вывода.

public void PrintMessage() {

 string myMessage = "Привет.";

 Consolе.WriteLine(myMessage);

}

Если рассмотреть результат трансляции этого метода компилятором C# в термины CIL, вы сразу заметите, что метод PrintMessage() определяет ячейку хранения для локальной переменной, используя директиву.locals. Локальная строка затем загружается и сохраняется в этой локальной неременной с помощью кодов операций ldstr (загрузка строки) и stloc.0 (это можно прочитать, как "запомнить текущее значение в локальной переменной с индексом 0").

Значение (снова с индексом 0) затем загружается в память с помощью кода операции ldloc.0 ("загрузить локальный аргумент с индексом 0") для использования в вызове метода System.Console.WriteLine() (указанном с помощью кода операции call). Наконец, происходит возврат из функции через код операции ret.

.method public hidebysig instance void PrintMessage() cil managed {

 .maxstack 1

**// Определение локальной строковой переменной (с индексом 0).**

 .locals init ([0] string myMessage)

**// Загрузка строки со значением "Привет."**

 ldstr "Привет."

**// Сохранение строкового значения в стеке в локальной переменной.**

 stloc.0

**// Загрузка значения с индексом 0.**

 ldloc.0

**// Вызов метода с текущим значением.**

 call void [mscorlib]System.Console::WriteLine(string)

 ret

}

**Замечание.**В программном коде CIL поддерживаются комментарии, использующие синтаксис двойной косой черты (а также синтаксис */\*…\*/).*Как и в C#, компилятором CIL комментарии просто игнорируются.

**Челночная технология разработки**

Вы уже знаете, как использовать ildasm.exe для просмотра программного кода CIL, генерируемого компилятором C#. Однако вы можете не знать о том, что ildasm.exe позволяет записать CIL-код, содержащийся в загруженном компоновочном блоке, во внешний файл. Имея программный код CIL в своем распоряжении, вы можете отредактировать и с помощью ildasm.exe – компилятора CIL – скомпилировать базовый код вновь.

Формально такой подход называется *челночной технологией разработки,*и эта технология может оказаться полезной в следующих случаях.

• Перед вами стоит задача изменить компоновочный блок, для которого нет исходного кода.

• Ввиду несовершенства компилятора языка .NET, сгенерировавшего неэффективный программный код CIL, вы хотите изменить этот код.

• Вы создаете компоновочные блоки, взаимодействующие в рамках COM, и вам приходится принимать во внимание то, что некоторые атрибуты IDL (Interface Definition Language – язык описания интерфейса) в процессе преобразования могут теряться (например, COM-атрибут [helpstring]).

Для примера использования челночной технологии разработки создайте новый файл (HelloProgram.cs) исходного кода C# с помощью обычного текстового редактора и определите в этом файле следующий тип класса (можете, конечно, использовать и Visual Studio 2005, но тогда не забудьте удалить файл AssemblyInfo.cs, чтобы уменьшить объем генерируемого CIL-кода).

**// Простое консольное приложение на языке C#.**

using System;

class Program {

 static void Main(string[] args) {

  Console.WriteLine("Hello CIL code!");

  Console.ReadLine();

 }

}

Сохраните этот файл в подходящем месте на своем диске и скомпилируйте его с помощью программы csc.exe.

csc HelloProgram.cs

Теперь откройте полученный файл HelloProgram.exe с помощью ildasm.exe и, используя опцию меню File→Dump, сохраните "сырой" программный код CIL в новом файле \*.il (HelloProgram.il) На вашем жестком диске (значения, предлагаемые в появляющемся диалоговом окне, вполне подойдут для наших целей). Теперь вы можете рассмотреть этот файл, используя любой текстовый редактор. Вот слегка откорректированный в снабженный некоторыми комментариями результат.

**// Компоновочные блоки, на которые мы ссылаемся.**

.assembly extern mscorlib {

 .publickeytoken = (В7 7A 5С 56 19 34 Е0 89)

 .ver 2:0:0:0

}

**// Ваш компоновочный блок.**

.assembly HelloProgram {

 .hash algorithm 0х00008004

.ver 0:0:0:0

}

.module HelloProgram.exe

.imagebase 0x00400000

.file alignment 0x00000200

.stackreserve 0x00100000

.subsystem 0x0003

.corflags 0x00000001

**// Определение класса Program.**

.class private auto ansi beforefieldinit Program extends [mscorlib]System.Object {

**.**method private hidebysig static void Main(string[] args) cil managed {

**// Обозначение этого метода, как точки входа**

**// выполняемого файла.**

  .entrypoint

  .maxstack.8

  IL\_0000: nop

  IL\_0001: ldstr "Hello CIL code!"

  IL\_0006: call void [mscorlib]System.Console::WriteLine(string)

  IL\_000b: nop

  IL\_000c: call string [mscorlib]System.Console::ReadLine()

  IL\_0011: pop

  IL\_0012: ret

 }

**// Конструктор, заданный по умолчанию.**

 .method public hidebysig specialname rtspecialname instance void .ctor() cil managed {

  .maxstack 8

  IL\_0000: ldarg.0

  IL\_0001: call instance void [mscorlib]System.Object::.ctor()

  IL\_0006: ret

 }

}

Во-первых, обратите внимание на то, что файл \*.il начинается с объявления всех внешних компоновочных блоков, на которые ссылается данный компоновочный блок. Здесь вы видите только одну директиву .assembly extern для одного обязательно присутствующего mscorlib.dll. Если бы ваша библиотека классов использовала типы из других внешних компоновочных блоков, вы бы обнаружили дополнительные директивы .assembly extern.

Далее следует формальное определение вашего компоновочного блока HelloProgram.exe, для которого указана версия 0.0.0.0, назначаемая по умолчанию (если вы не укажете иное значение с помощью атрибута [AssemblyVersion]). После этого приводятся другие описания компоновочного блока, для которых используются другие директивы CIL (такие, как .module, .imagebase и т.д.).

После указания ссылок на внешние компоновочные блоки и определения текущего компоновочного блока идет определение типа Program. Обратите внимание на то, что директива.class имеет несколько атрибутов (многие из которых необязательны), – например, атрибут extends, задающий базовый класс типа.

.class private auto ansi beforefieldinit Program **extends**[mscorlib]System.Object {…}

Большой кусок программного кода CIL соответствует конструктору класса, заданному по умолчанию, и методу Main(). Оба они определены (в частности) с помощью директивы .method. После определения этих членов с помощью подходящих директив и атрибутов они реализуются с помощью различных кодов операций.

Важно понять, что в CIL при взаимодействии с типами .NET (например, с System.Console) *всегда*необходимо использовать абсолютные имена типов. Более того, к абсолютному имени типа *всегда*должен добавляться (в квадратных скобках) префикс с понятным именем компоновочного блока, определяющего этот тип. Взгляните на CIL-реализацию Main().

.method private hidebysig static void Main(string[] args) cil managed {

 .entrypoint

 .maxstack 8

 IL\_0000: nop

 IL\_0001: ldstr "Hello CIL code!"

 IL\_0006: call void [mscorlib]System.Console::WriteLine(string)

 IL\_000b: nop

 IL\_000c: call string [mscorlib]System.Console::ReadLine()

 IL\_0011: pop

 IL\_0012: ret

}

Реализация конструктора, заданного по умолчанию, в терминах программного кода CIL включает еще одну относящуюся к загрузке инструкцию (ldarg.0). В данном случае значение загружается в стек не как пользовательская переменная, указанная нами, а как текущая объектная ссылка (подробности этого процесса будут описаны позже). Также обратите внимание на то, что конструктор, заданный по умолчанию, явно вызывает конструктор базового класса.

.method public hidebysig specialname rtspecialname instance void .ctor cil managed {

 .maxstack 8

 IL\_0000: **ldarg.0**

 IL\_0001: **call**instance void [mscorlib]System.Object::.ctor()

 IL\_0006: **ret**

}

**Роль меток в программном коде CIL**

Вы. конечно, заметили, что в каждой строке программного кода реализации содержится префикс в форме лексемы IL\_XXX: (например, IL\_0000: IL\_0001: и т.д.). Эти лексемы называются *метками кода,* и они могут иметь любой вид, какой вы только пожелаете (лишь бы они не дублировались в пределах одного и того же контекста). При записи содержимого компоновочного блока в файл с помощью ildasm.exe автоматически генерируются метки кода, имеющие вид IL\_XXX:. Но вы можете изменить их с тем, чтобы они стали более информативными.

.method private hidebysig static void Main(string[] args) cil managed {

 .entrypoint

 .maxstack 8

**Nothing\_1**: nop

**Load\_String:**ldstr "Hello CIL code!"

**PrintToConsole:**call void **[mscorlib]System.Console::WriteLine(string)**

**Nothing\_2:**nop

**WaitFor\_KeyPress:**call string **[mscorlib] System.Console::ReadLine()**

**RemoveValueFromStack**: pop

**Leave\_Functlon**: ret

}

Суть в том, что большинство меток кода совсем необязательно. Единственным случаем, когда метки кода оказываются по-настоящему полезными (и обязательными), является случай, когда в программном коде CIL используются ветвления или циклические конструкции. Например, в нашем случае вы можете исключить метки вообще.

.method private hidebysig static void Main(string[] args) cil managed {

 .entrypoint

 .maxstack 8

 nop

 ldstr "Hello CIL code!"

 call void **[mscorlib]System.Console::WriteLine(string)**

 nop

 call string **[mscorlib]System.Console::ReadLine()**

 pop

 ret

}

**Взаимодействие с CIL: модификация файла \*.il**

Теперь, когда вы понимаете, как компонуется базовый файл CIL, давайте завершим наш эксперимент с челночной технологией разработки программ. С помощью изменения CIL-кода в файле \*.il мы должны выполнить следующее.

• Добавить ссылку на компоновочный блок System.Windows.Forms.dll.

• Загрузить локальную строку в Main().

• Вызвать метод System.Windows.Forms.MessageBox.Show(), используя локальную строковую переменную в качестве его аргумента.

Первым шагом является добавление новой директивы.assembly (с атрибутом extern), которая укажет, что используется System.Windows.Forms.dll. Для этого просто добавьте в файл \*.il следующую программную логику после ссылки на внешний компоновочный блок mscorlib.

.assembly extern System.Windows.Forms {

 .publickeytoken = (B7 7A 5C 56 19 34 E0 89)

 .ver 2:0:0:0

}

Значение, указанное директивой .ver. может у вас оказаться другим, поскольку оно зависит от версии платформы .NET, установленной на вашей машине. Здесь указано использование System.Windows.Forms.dll версии 2.0.0.0 с кодом открытого ключа В77А5С561934Е089. Если открыть GAC (см. главу 11) и найти там компоновочный блок System.Windows.Forms.dll, можно скопировать правильный номер версии и значение открытого ключа со страницы свойств этого компоновочного блока.

.method private hidebysig static void Main(string[] args) cil managed {

 .entrypoint

 .maxstack 8

**// Задача: написать новый CIL-код.**

}

Итак, целью является помещение новой строки в стек и вызов метода MessageBox.Show() (а не метода Console.WriteLine()). Напомним, что при указании имени внешнего типа следует использовать абсолютное имя типа (в совокупности с понятным именем компоновочного блока). С учетом этого обновите метод Main() так, как показано ниже.

.method private hidebysig static void Main(string[] args) cil managed {

 .entrypoint

 .maxstack 8

**ldstr**"CIL работает прекрасно!"

**call**valuetype [System.Windows.Forms] System.Windows.Forms.DialogResult [System*.*Windows.Forms] System.Windows.Forms.MessageBox::Show(string)

**pop**

**ret**

}

В результате вы получите программный код CIL, соответствующий следующему определению класса C#.

public class Program {

 static void Main(string[] args) {

  System.Windows.Forms.MessageBox.Show("CIL работает прекрасно!");

 }

}

**Компиляция CIL-кода с помощью ilasm.exe**

Сохранив измененный файл \*.il, вы можете скомпилировать новый компоновочный блок .NET, используя для этого утилиту ilasm.exe (компилятор CIL). Возможно, вы удивитесь тому, что компилятор CIL имеет гораздо меньше опций командной строки, чем компилятор C#. В табл. 15.1 приводятся их описания.

Таблица 15.1. Опции командной строки ilasm.exe

|  |  |
| --- | --- |
| **Опция** | **Описание** |
| /debug | Включает информацию отладки (такую как имена локальных переменных и аргументов, а также номера строк) |
| /dll | Создает выходной файл" \*.dll |
| /exe | Создает выходной файл *\*.*exe. Это значение устанавливается по умолчанию, поэтому его можно опустить |
| /key | Компилирует компоновочный блок со строгим именем, используя заданный файл \*.snk |
| /noautoinherit | Запрещает автоматическое наследование типов класса из System. Object, когда конкретный базовый класс не определен |
| /output | Указывает имя и расширение выходного файла. Если флаг /output не используется, имя выходного файла будет соответствовать имени первого исходного файла |

Чтобы откомпилировать обновленный файл simplehelloclass.il в .NET-файл \*.exe, в командном окне Visual Studio 2005 выполните следующую команду.

ilasm.exe HelloProgram.il

Если все пройдет без сбоев, вы должны получить вывод, подобный показанному на рис. 15.1.

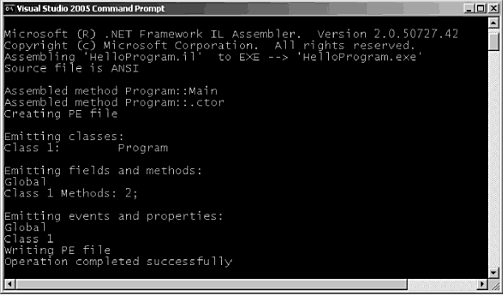


Рис. 15.1. Компиляция файлов \*.il с помощью ilasm.exe

После этого вы сможете выполнить свое новое приложение. Достаточно очевидно, что теперь вместо сообщения в окне консоли вы должны увидеть окно Windows с вашим сообщением (рис. 15.2).



Рис. 15.2. Результат челночной технологии

**Компиляция CIL-кода с помощью SharpDevelop**

Для работы с файлами \*.il вы можете использовать бесплатную среду разработки SharpDevelop (см. главу 2). При создании нового "комбината" (для этого выберите File→New Combine из меню), одним из вариантов выбора является создание рабочего пространства CIL-проекта. Хотя SharpDevelop пока что не предлагает поддержку IntelliSense для CIL-проектов, лексемы CIL выделяются цветом, и вы получаете возможность компилировать и выполнять свои приложения непосредственно в окне IDE (а не в командной строке, как в случае ilasm.exe).

**Компиляция CIL-кода с помощью ILIDE#**

Если вам интересно поэкспериментировать с языком программирования CIL, рекомендую загрузить самую последнюю версию бесплатного редактора исходных текстов CIL, имеющего открытый код и название ILIDE#. Этот инструмент, подобно SharpDevelop, обеспечивает выделение цветом ключевых слов и программных структур, интеграцию с ilasm.exe и набор соответствующих инструментов. В отличие от SharpDevelop, последняя версия ILIDE# поддерживает IntelliSense для CIL. Установщик ILIDE# можно загрузить со страницы <http://ilide.aspfreeserver.com/default-en.aspx> (этот адрес URL может измениться). На рис. 15.3 показано окно ILIDE# в действии.

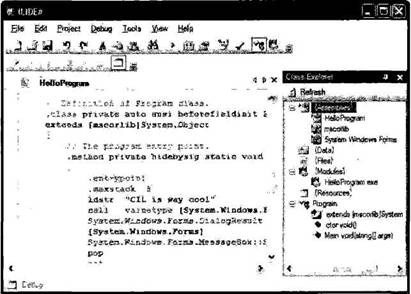


Рис. 15.3. Редактор ILIDE# – бесплатная среда разработки для CIL

**Роль peverify.exe**

При создании или модификации компоновочных блоков, в которых используется программный код CIL, всегда целесообразно проверить, будет ли скомпилированный двоичный образ правильно сформирован с точки зрения правил .NET. Для этого можно использовать средство командной строки peverify.exe*.*

peverifу HelloProgram.exe

Этот инструмент проверит все коды операций в указанном компоновочном блоке на соответствие правилам CIL. Например, в терминах CIL-кода стек оценок должен всегда опустошаться: перед выходом из функции, Если вы забудете извлечь из него какие-то значения, компилятор ilasm.exe все равно сгенерирует допустимый компоновочный блок (поскольку компиляторы "заботятся" только о *синтаксисе).*

А вот peverifу.exe, с другой стороны, заботится о семантике. Если вы забудете очистить стек перед выходом из функции, peverify.exe сообщит вам об этом.

**Исходный код**. Файл HelloProgram.il размещен в подкаталоге, соответствующем главе 15.

**Директивы и атрибуты CIL**

Теперь, когда вы знаете, как использовать ildasm.exe и ilasm.exe в рамках челночной технологии разработки, мы можем заняться непосредственным анализом синтаксиса и семантики CIL. Следующие разделы предлагают описание процесса построения пользовательского пространства имен, содержащего определенный набор типов. Чтобы упростить рассмотрение, эти типы не будут содержать никакого программного кода реализации их членов. После того как вы поймете, как создаются пустые типы, вы сможете сосредоточить все свое внимание на процессе создания "реальных" членов типа с помощью кодов операций CIL.

**Ссылки на внешние компоновочные блоки**

С помощью любого редактора создайте новый файл, назвав его CilTypes.il. Сначала вы должны указать список внешних компоновочных блоков, используемых текущим компоновочным блоком (в нашем примере мы будем использовать только типы из mscorlib.dll). Для этого нужно указать директиву .assembly с атрибутом external. При ссылке на строго именованный компоновочный блок, такой как mscorlib.dll, вы должны также указать директивы .publickeytoken и .ver.

.assembly extern mscorlib {

 .**publickeytoken** = (B7 7A 5C 56 19 34 E0 89)

 .**ver** 2:0:0:0

}

**Замечание.**Строго говоря, явная ссылка на внешний компоновочный блок mscorlib.dll не является обязательной, поскольку ilasm.exe добавит такую ссылку автоматически.

**Определение текущего компоновочного блока**

Следующей задачей является определение компоновочного блока, который вы хотите построить. Это делается с помощью директивы .assembly. В простейшем случае компоновочный блок можно определить с помощью простого указания понятного имени соответствующего двоичного файла.

**// Наш компоновочный блок.**

.assembly CILTypes {}

Это действительно определяет новый компоновочный блок .NET, но обычно в рамках декларации компоновочного блока размещаются дополнительные директивы. Для нашего примера добавьте в определение компоновочного блока номер версии 1.0.0.0 с помощью директивы .ver (заметьте, что все числовые идентификаторы в определении должны разделяться *двоеточием,*не точкой, как в C#).

**// Наш компоновочный блок.**

.assembly CILTypes {

**.ver 1:0:0:0**

}

Поскольку компоновочный блок CILTypes является одномодульным компоновочным блоком, определение этого компоновочного блока завершается единственной директивой.module, которая указывает официальное имя двоичного .NET-файла, CILTypes.dll.

.assembly CILTypes {

**.ver 1:0:0:0**

}

**// Этот модуль является одномодульным компоновочным блоком.**

**.module**CILTypes.dll

Кроме директив .assembly и .module, есть и другие CIL-директивы, обеспечивающие дальнейшее уточнение структуры создаваемого двоичного файла .NET. В табл. 15.2 предлагаются описания еще двух директив уровня компоновочного блока,

**Таблица 15.2.**Дополнительные директивы компоновочного блока

|  |  |
| --- | --- |
| **Директива** | **Описание** |
| .mresources | Если компоновочный блок использует встраиваемый ресурс (например, точечный рисунок или таблицу строк), эта директива используется для идентификации имени файла, содержавшего такой ресурс. В главе 20 ресурсы .NET рассматриваются подробно |
| .subsystem | Эта директива CIL используется для указания предпочтительного пользовательского интерфейса для выполнения компоновочного блока, например, значение 2 означает, что компоновочный блок должен выполняться в рамках графического интерфейса с поддержкой форм, а значение 3 означает консольное приложение |

**Определение пространств имен**

Итак, вы определили вид своего компоновочного блока (и необходимые внешние ссылки). Теперь можно создать пространство имен .NET (МуNamespace), используя для этого директиву .namespace.

**// Наш компоновочный блок имеет одно пространство имен.**

.**namespace** MyNamespace {}

Как и в C#, определение пространства имен CIL можно вложить во внешнее пространство имен. Вот пример вложения нашего пространства имен в корневое пространство имен с именем IntertechTraining.

.namespace IntertechTraining {

 .**namespace**MyNamespace {}

}

Кроме того, как и C#, язык CIL позволяет определить вложенное пространство имен так.

// **Определение вложенного пространства имен.**

.namespace IntertechTraining.MyNamespace{}

**Определение типов класса**

Пустые пространства имен не представляют собой большого интереса, поэтому давайте выясним, как в CIL определяется тип класса. Вполне логично, что для этого используется директива .class. Однако эта простая директива может иметь множество дополнительных атрибутов, уточняющих природу создаваемого типа. Для примера мы добавим простой общедоступный класс с именем MyBaseClass. Как и в C#, если не указать базовый класс явно, соответствующий тип будет автоматически получаться из System.Object.

.namespace MyNamespace {

**// В качестве базового класса предполагается System.Object.**

**.class**public MyBaseClass {}

}

Для создания типа класса, являющегося производным от любого класса, отличного от System.Object, используется атрибут extends. При ссылке на тип, определенный в пределах того же компоновочного блока, CIL требует, чтобы вы указали абсолютное имя (однако для базового класса из того же компоновочного блока вы можете опустить префикс, представляющий понятное имя компоновочного блока). Так, следующий вариант модификации MyBaseClass приведет к ошибке компиляции.

**// Это компилироваться не будет!**

.namespace MyNamespace {

 .class public MyBaseClass {}

 .class public MyDerivedClass **extends MyBaseClass {}**

}

Чтобы корректно определить родительский класс для MyDerivedClass, следует указать полное имя MyBaseClass, как показано ниже.

**// Так будет лучше!**

.namespace MyNamespace {

 .class public MyBaseClass {}

 .class public MyDerivedClass **extends MyNamespace.MyBaseClass {}**

}

Вдобавок к атрибутам public и extends определение класса CIL может иметь множество дополнительных спецификаторов, задающих параметры видимости типа, размещения полей и т.д. В табл. 15.3 предлагаются описаний некоторых атрибутов, которые могут использоваться с директивой .class.

**Таблица 15.3.**Атрибуты, которые могут использоваться с директивой .class

|  |  |
| --- | --- |
| **Атрибут** | **Описание** |
| public, private, nested assembly, nested famandassem, nested family, nested famorassem, nested public, nested private | В CIL определяется множество атрибутов, используемых для указании параметров видимости типа. Как видите, CIL предлагает целый ряд возможностей, не доступных в C# |
| abstract sealed | Эти два атрибута можно добавить к директиве.class, чтобы определить абстрактный класс или изолированный класс, соответственно |
| auto sequential explicit | Эти атрибуты используются для информирования CLR о том, как следует размещать поля данных в памяти. Для типов класса вполне подойдет вариант, используемый по умолчанию (auto) |
| extends implements | Эти атрибуты позволяют определить базовый класс типа (с помощью extends) или реализовать интерфейс (с помощью implements) |

**Определение и реализация интерфейсов**

Как бы это странно ни выглядело, типы интерфейса определяются в CIL с помощью директивы .class. Но когда директива .сlass сопровождается атрибутом interface, соответствующий тип реализуется, как тип интерфейса CTS (Common Type System – общая система типов). После определения интерфейс можно привязать к типу класса или структуры с помощью CIL-атрибута implements.

.namespace MyNamespace {

**// Определение интерфейса.**

 .class public **interface**IMyInterface {}

 .class public MyBaseClass {}

**// Теперь DerivedTestClass реализует IAmAnInterface.**

 class public MyDerivedClass

  extends MyNamespace.MyBaseClass **implements**MyNamespace.IMyInterface {}

}

Как было показано в главе 7, интерфейсы могут выступать в качестве базовых интерфейсов в отношении других типов интерфейса, в результате чего создаются иерархии интерфейсов. Однако, вопреки вашим возможным догадкам, атрибут extends нельзя использовать для получения интерфейса А из интерфейса В. Атрибут extends используется только для указания базового класса типа. Чтобы расширить интерфейс, вы должны еще раз использовать атрибут implements.

**// Расширение интерфейсов в терминах CIL.**

.class public interface IMyInterface {}

.class public interface IMyOtherInterface **implements**MyNamespace.IMyInterface {}

**Определение структур**

Директива .class может использоваться и для определения CTS-структуры, если соответствующий тип расширяет System.ValueType. Кроме того, такая директива .class сопровождается атрибутом sealed (поскольку структура не может быть базовой по отношению к другим типам, характеризуемым значениями). Если вы попытаетесь сделать иначе, ilasm.exe сгенерирует ошибку компиляции.

**// Структура всегда должна быть изолированной**.

.class public **sealed**MyStruct extends [mscorlib]**System.ValueType**{}

Полезно знать о том, что CIL предлагает специальное сокращение для определения типа структуры. Если вы используете атрибут value, новый тип будет производным от [mscorlib] System.ValueType и получит атрибут sealed автоматически. Таким образом, можно определить MyStruct так.

**// Сокращенная запись для определения структуры.**

.class public **value**MyStruct{}

**Определение перечней**

Перечни .NET (как вы помните) получаются из класса System.Enum, производного от System.ValueType (и, таким образом, тоже должны быть изолированными). Чтобы определить перечень в терминах CIL, следует просто расширить [mscorlib]System.Enum.

**// Перечень.**

.class public **sealed**MyEnum extends [mscorlib]**System.Enum**{}

Как и для структур, для определения перечней имеется специальное сокращение, атрибут enum.

**// Сокращенная запись для определения перечня.**

.class public **enum**MyEnum {}

**Замечание.**Последний из фундаментальных типов данных .NET, делегат, тоже имеет специальное представление в CIL. Подробности можно найти в главе 6.

**Компиляция файла CILTypes.il**

Даже если вы не добавите никаких членов или иного программного кода в определенные вами типы, вы можете скомпилировать файл \*.il в компоновочный блок DLL (иное просто невозможно, поскольку вы не указали метод Main()).

Откройте окно командной строки и введите следующую команду.

ilasm /dll CilTypes.il

После этого вы сможете открыть свой двоичный файл в ildasm.exe (рис. 15.4).

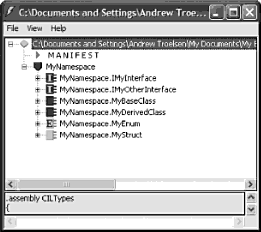


Рис. 15.4. Содержимое компоновочного блока CILTypes.dll

Проверив содержимое своего компоновочного блока, запустите для него peverify.exe. В результате будет выдан целый ряд сообщений об ошибках, поскольку все ваши типы пусты. Чтобы понять, как заполнить типы содержимым, мы должны сначала рассмотреть базовые типы данных CIL.

**Исходный код**. Файл CilTypes.il размещен в подкаталоге, соответствующем главе 15.

**Соответствие между типами библиотеки базовых классов .NET, C# и CIL**

В табл. 15.4 показано соответствие между типами базовых классов .NET и ключевыми словами C#, а также между ключевыми словами C# и командами CIL. Там же представлены сокращенные обозначения констант, используемые для CIL-типов. Чуть позже вы сможете убедиться в том, что при использовании кодов операций C#, часто используются и ссылки на эти константы.

**Определение членов типов в CIL**

Вы уже знаете, что типы .NET могут определить различные члены. Перечни содержат некоторый набор пар имен и значений. Структуры и классы могут иметь конструкторы, поля, методы, свойства, статические члены и т.д. В предыдущих 14 главах вы уже могли видеть фрагменты определений CIL для таких элементов, но тем не менее, ниже предлагается краткая сводка того, как различные члены отображаются в примитивы CIL.

**Таблица 15.4.**Связь между типами базовых классов .NET и ключевыми словами C#, а также их проекция в CIL

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип базового класса .NET** | **Ключевое слово C#** | **Представление CIL** | **Обозначение для константы CIL** |
| System.SByte | sbyte | int8 | I1 |
| System.Byte | byte | unsigned int8 | U1 |
| System.Int16 | short | int16 | I2 |
| System.UInt16 | ushort | unsigned int16 | U2 |
| System.Int32 | int | int32 | I4 |
| System.UInt32 | uint unsigned | int32 | U4 |
| System.Int64 | long | int64 | I8 |
| System.UInt64 | ulong | unsigned int64 | U8 |
| System.Char | char | char | CHAR |
| System.Single | float | float32 | R4 |
| System.Double | double | float64 | R8 |
| System.Boolean | bool | bool | BOOLEAN |
| System.String | string | string | – |
| System.Object | object | object | – |
| System.Void | void | void | VOID |

**Определение полей данных**

Перечни, структуры и классы могут поддерживать поля данных. Во всех случаях для указания таких полей используется директива. field. Например, чтобы добавить немного сути в каркас MyEnum, давайте определим для него три пары имен и значений (заметьте, что значения здесь указываются в скобках).

.class public auto ansi sealed MyEnum extends [mscorlib]System.Enum {

**.field**public **static literal**valuetype MyNamespace.MyEnum NameOne = int32(0)

**.field**public **static literal**valuetype MyNamespace.MyEnum NameTwo = int32(1)

**.field**public **static literal**valuetype MyNamespace.MyEnum NameThree = int32(2)

}

Поля, размещаемые в рамках контекста типа .NET, производного от System.Enum, сопровождаются атрибутами static и literal. Вам должно быть ясно, что эти атрибуты соответствуют полям данных, имеющим фиксированное значение и доступным из данного типа непосредственно (например, с помощью MyEnum.NameOne).

**Замечание**. Значения, присваиваемые полям перечня, могут также быть шестнадцатиричными.

Конечно, при определении полей данных в пределах класса или структуры вы не ограничены использованием только открытых статических литералов. Можно, например, добавить в MyBaseClass поддержку двух приватных полей данных уровня экземпляра.

.class public MyBaseClass {

**.field private**string stringField

 .**field private**int32 intField

}

Как и в C#, полям данных класса будут автоматически назначены подходящие значения для непользовании по умолчанию. Чтобы позволить пользователю объекта указать во время создания объекта пользовательские значения для приватных полей данных, придется (конечно) создать пользовательские конструкторы.

**Определение конструкторов типов**

Система CTS (общая система типов) поддерживает конструкторы как уровня экземпляра, так и уровня класса (статические конструкторы). В терминах CIL для конструкторов уровня экземпляра используется лексема .ctor, а для статических конструкторов – лексема .cctor (конструктор класса). Обе эти лексемы CIL должны сопровождаться атрибутами rtspecialname (специальное имя возвращаемого типа) и specialname. Эти атрибуты используются для идентификации специальных лексем CIL, позволяющих уникальное толкование в каждом языке .NET. Например, в C# конструкторы не определяют возвращаемый тип, однако в терминах CIL возвращаемым значением конструктора на самом деле будет void.

.class public MyBaseClass {

 .field private string stringField

 .field private int32 intField

 .method public hidebysig **specialname rtspecialname instance**void .**ctor**(string s, int32 i) cil managed {

**// Задача: добавить необходимый программный код.**

 }

}

Обратите внимание на то, что директива .ctor сопровождается атрибутом instance (поскольку это не статический конструктор). Атрибуты cil managed означают, что в контексте этого метода содержится программный код CIL (а не программный код, не являющийся управляемым), который может использоваться в межплатформенных запросах.

**Определение свойств**

Свойства и методы также имеют специальные представления в CIL. Чтобы в нашем примере обеспечить в MyBaseClass поддержку открытого свойства TheString, можно использовать следующий CIL-код (заметьте, что здесь опять используется атрибут specialname).

.class public MyBaseClass {

 …

**.method**public hidebysig **specialname**instance string get\_TheString() cil managed {

**// Задача: добавить необходимый программный код…**

 }

**.method**public hidebysig **specialname**instance void set\_TheString(string 'value') cil managed {

**// Задача: добавить необходимый программный ход.…**

 }

**.property**instance string TheString() {

**.get**instance string MyNamespace.MyBaseClass::get\_TheString()

**.set**instance void MyNamespace.MyBaseClass::set\_TheString(string)

 }

}

Напомним, что в терминах CIL свойства будут представлены парой методов, имеющих префиксы get\_ и set\_. Директива .property использует соответствующие директивы .get и .set, чтобы связать синтаксис свойства со "специально именованными" методами.

**Замечание.**Указанные выше определения свойств компилироваться не будут, поскольку пока что не реализована сама логика чтения и модификации данных.

**Определение параметров членов**

Теперь предположим, что нужно определить методы, имеющие аргументы. По сути, указание аргументов в CIL (приблизительно) соответствует аналогичной операции в C#. Например, аргумент определяется с помощью указания типа данных после имени соответствующего параметра. К тому же, как и в C#, в CIL обеспечиваются возможности ввода, вывода и передачи параметров по ссылке. Также в CIL позволяется определять аргумент массива параметров (в C# это делается с помощью ключевого слова params) и необязательные параметры (которые в C# не поддерживаются, но допускаются в VB .NET).

Чтобы показать пример определения параметров непосредственно в CIL, предположим, что нам нужно построить метод, который получает int32 (по значению), int32 (по ссылке), [mscorlib] System.Collections.ArrayList и имеет единственный выходной параметр (типа int32). В терминах C# этот метод должен выглядеть приблизительно так.

public static void MyMethod(int inputInt, **ref**int refInt, ArrayList ar, **out**int outputInt) {

 outputInt = 0; **// Просто чтобы удовлетворить компилятор C#…**

}

Если спроецировать этот метод в CIL-код, вы обнаружите, что ссылки на параметры C# будут обозначены знаком амперсанда (&), добавленного в виде суффикса к типу данных, соответствующему параметру (int32&). Для выходных параметров тоже используется суффикс &, но, кроме того, они обозначены маркером CIL [out], Также обратите внимание на то, что в том случае, когда параметр является ссылочным типом (как тип [mscorlib]System.Collections.ArrayList в нашем примере), ему предшествует лексема class (не путайте с директивой .class!).

.method public hidebysig static void MyMethod(int32 inputInt, **int32&**refInt, **class**[mscorlib]System.Collections.ArrayList ar, **[out] int32&**outputInt) cil managed {

 …

}

**Анализ кодов операций CIL**

Заключительной темой нашего обсуждения в этой главе в отношении программного кода CIL будет роль кодов операций. Напомним, что код операции – это просто лексема CIL, используемая для построения логики реализации данного члена. Полный набор кодов операций CIL (который сам по себе довольно велик) можно разбить на следующие большие категории.

• Коды операций для управления программой

• Коды операций для оценки выражений

• Коды операций для осуществления доступа к значениям в памяти (через параметры, локальные переменный и т.п.)

Чтобы продемонстрировать некоторые возможности реализации членов средствами CIL, в табл. 15.5 предлагаются описания некоторых из наиболее часто используемых кодов операций, непосредственно связанных с логикой реализации членов. Кроме того, коды операций в данной таблице сгруппированы по функциональности.

**Таблица 15.5.**Коды операций CIL, связанные с реализацией членов

|  |  |
| --- | --- |
| **Коды операций** | **Описание** |
| add, sub, mul, div, rem | Позволяют выполнять сложение, вычитание, умножение и деление для пар значений (rem возвращает остаток от деления) |
| and, or, not, xor | Позволяют выполнять соответствующие бинарные операции для пар значений |
| ceq, cgt, clt | Позволяют сравнивать пару значений из стека различными способами, например: ceq: сравнение в отношении равенства cgt: сравнение в отношении "больше" clt: сравнение в отношении "меньше" |
| box, unbox | Используются для конвертирования ссылочных типов и типов, характеризуемых значениями |
| ret | Используется для выхода из метода и (если это необходимо) возвращения значения вызывающей стороне |
| beq, bgt, ble, blt, switch | Используются (в дополнение к множеству других родственных кодов операций) для управления логикой ветвления в методах, например: beq: переход к заданной метке, если выполняется равенство bgt: переход к заданной метке, если больше ble: переход к заданной метке, если меньше или равно blt: переход к заданной метке, если меньше Все коды операций, связанные с ветвлением, требуют указания метки CIL-кода, по которой должен осуществляться переход в том случае, когда соответствующее сравнение возвращает true |
| call | Используется для вызова члена указанного типа |
| newarr, newobj | Позволяет разместить в памяти новый массив или новый объект (cоответственно) |

Следующая большая категория кодов операций CIL (подмножество которой показано в табл. 15.6) используется для загрузки аргументов в виртуальный стек выполнения. Обратите внимание на то, что эти относящиеся к загрузке коды операций имеют префикс ld (load – загрузка).

**Таблица 15.6.**Коды операций CIL для помещения данных в стек

|  |  |
| --- | --- |
| **Код операции** | **Описание** |
| ldarg (с множеством вариаций) | Помещает в стек аргумент метода. Вдобавок к общей операции ldarg (для которой требуется указать индекс, идентифицирующий аргумент), есть множество ее вариаций. Например, ldarg с числовым суффиксом (ldarg\_0) используется для загрузки соответствующего аргумента. Другие вариации ldarg позволяют с помощью кодов констант CIL из табл. 15.4 указать конкретный тип загружаемых данных (например, ldarg\_I4 для int32), а также тип данных и значение (ldarg\_I4\_5 для загрузки int32 со значением 5) |
| ldc (с множеством вариаций) | Помещает в стек значение константы |
| ldfld (с множеством вариаций) | Помещает в стек значение поля уровня экземпляра |
| ldloc (с множеством вариаций) | Помещает в стек значение локальной переменной |
| ldobj | Читает все значения объекта, размещенного в динамической памяти, и помещает их в стек |
| ldstr | Помещает в стек строковое значение |

Вдобавок к множеству специальных кодов операций загрузки, CIL предлагает набор кодов операций, которые непосредственно "выталкивают" из стека самое верхнее значение. Как продемонстрировали первые несколько примеров этой главы, удаление значения из стека обычно выполняется с целью последующего сохранения этого значения в локальной памяти для дальнейшего использования (например, в качестве параметра при последующем вызове метода). С учетом этого становится ясно, почему многие коды операций, связанные с удалением текущего значения из виртуального стека выполнения, имеют префикс st (store – сохранять). Соответствующие описания приведены в табл. 15.7.

**Таблица 15.7**. Коды операций для извлечения данных из cтека

|  |  |
| --- | --- |
| **Код операции** | **Описание** |
| pop | Удаляет значение, находящееся в настоящий момент на вершине стека, но не обеспечивает сохранение этого значения |
| starg | Сохраняет значение из вершины стека в аргументе метода с указанным индексом |
| stloc (c множеством вариаций) | Удаляет значение, находящееся на вершине стека, и запоминает это значение в переменной с указанным индексом из списка локальных переменных |
| stobj | Копирует значение указанного типа из стека в память по указанному адресу |
| stsfld | Заменяет значение статического поля значением из cтека |

Следует также знать о том, что различные коды операций CIL при выполнении своих задач *неявно*удаляют значения из стека. Например, при вычитании одного числа из другого с помощью операции sub следует учитывать то, что прежде чем выполнить соответствующее вычисление, sub "вытолкнет" из стека два доступных значения. После выполнения операции в стек добавляется результат (как неожиданно!).

**Директива .maxstack**

При реализации метода непосредственно средствами CIL нужно помнить о специальной директиве, которая называется .maxstack. Как следует из ее названия, директива .maxstack задает максимальное число переменных, которые может вместить стек в любой момент времени при выполнении метода. К счастью, директива .maxstack имеет значение по умолчанию (8), которого оказывается достаточно для подавляющего большинства методов, создаваемых разработчиками. Но у вас также есть возможность определить это значение явно, чтобы при желании вручную указать числа локальных переменных в стеке.

.method public hidebysig instanсе void Speak() cil managed {

**// В контексте этого метода в стек помещается ровно**

**// одно значение (строковый литерал).**

 .maxstack 1

 ldstr "Всем привет…"

 call void [mscorlib]System.Consolr::WriteLine(string)

 ret

}

**Объявление локальных переменных**

Давайте выясним, как объявляются локальные переменные. Предположим, что мы должны построить в терминах CIL метод MyLocalVariables(), не имеющий никаких аргументов и возвращающий void. В этом методе мы должны определить три локальные переменные типов System.String, System.Int32 и System.Object. В C# соответствующий программный код мог бы выглядеть так, как показано ниже (напомним, что локальные переменные не получают значения по умолчанию, поэтому им перед использованием необходимо присвоить начальные значения).

public static void MyLocalVariables() {

 string myStr = "CIL me dude…";

 int myInt = 33;

 object myObj = new object();

}

Если создавать MyLocalVariables() непосредственно в CIL, можно было бы написать следующее,

.method public hidebysig static void MyLocalVariables() cil managed {

 .maxstack 6

**// Определение трех локальных переданных.**

**.locals init**(**[0]** string myStr, **[1]**int32 myInt, **[2]**object myObj)

**// Загрузка строки в виртуальный стек выполнения.**

ldstr "CIL me dude…"

**// Извлечение текущего значения и сохранение его**

**// в локальной переменной [0].**

 stloc.**0**

**// Загрузка константы типа 'i4'**

**// (сокращение для int32) со значением 33.**

 ldc.i4 33

**// Извлечение текущего значения и сохранение его**

**// в локальной переменной [1].**

 stloc.**1**

**// Создание нового объекта и помещение его в стек.**

newobj instance void [mscorlib]System.Object::.ctor()

**// Извлечение текущего значения и сохранение его**

**// в локальной переменной [2].**

 stloc.**2**

 ret

}

Как видите, в CIL при размещении локальных переменных сначала используется директива .locals с атрибутом init. При этом в скобках каждая переменная связывается со своим числовым индексом (здесь это [0], [1] и [2]). Каждый индекс идентифицируется типом данных и (необязательно) именем переменной. После определения локальных переменных соответствующее значение загружается в стек (с помощью подходящих кодов операций, связанных с загрузкой) и запоминается в локальной переменной (с помощью подходящих кодов операций для сохранения значений).

**Связывание параметров с локальными переменными**

Вы только что видели, как в CIL с помощью .local init объявляются локальные переменные, однако нужно еще выяснить, как передать отступающие параметры локальным методом. Рассмотрим следующий статический метод C#.

public **static**int Add(int **a***,*int **b**) {

 return a + b;

}

Этот внешне "невинный" метод в терминах CIL существенно более "многословен". Во-первых, поступающие аргументы (а и b) следует поместить в виртуальный стек выполнение с помощью кода операций ldarg (загрузка аргумента). Затем используется код операции add, чтобы извлечь два значения из стека, найти сумму и снова сохранить значение в стеке. Наконец, эта сумма извлекается из стена и возвращается вызывающей стороне с помощью кода операции ret. Если дизассемблировать указанный метод C# с помощью ildasm.exe, вы обнаружите, что компилятор csc.exe добавляет множество дополнительных лексем, хотя сущность CIL-кода оказывается исключительно простой.

.method public hidebysig static int32 Add(int32 **a**, int32 **b**) cil managed {

 .maxstack 2

**ldarg.0**// Загрузка 'a' в стек,

**ldarg.1**// Загрузка 'b' стек,

**add**// Сложение этих значений.

 ret

}

**Скрытая ссылка this**

Обратите внимание на то, что в рамках программного кода CIL для ссылок на два входных аргумента (а и b) используются их индексы позиции (индекс 0 и индекс 1, поскольку индексация в виртуальном стеке выполнения начинается с нуля).

При анализе программного кода и его создании непосредственно в CIL следует быть очень внимательным, поскольку каждый (нестатический) метод, имеющий входные аргументы, автоматически получает неявный дополнительный параметр, который является ссылкой на текущий объект (это должно вызвать аналогию с ключевым словом C# this). Поэтому, если определить метод Add(), как нестатический

**// Уже не является статическим!**

public int Add(int a, int b) {

 return a + b;

}

то входные аргументы а и b будут загружаться с помощью ldarg.1 и ldarg.2 (а не с помощью ожидаемых ldarg.0 и ldarg.1). Причина как раз в том, что ячейка 0 будет содержать неявную ссылку this. Рассмотрите следующий псевдокод.

**// Это только псевдокод!**

.method public hidebysig static int32 AddTwoIntParams(**MyClass\_HiddenThisPointer this,**int32 a, int32 b) cil managed {

 ldarg.0 // Загрузка **MyClass\_HiddenThisPointer**в стек,

 ldarg.1 // Загрузка 'а' в стек.

 ldarg.2 // Загрузка 'b' в стек.

 …

}

**Представление итерационных конструкций**

Итерационные конструкции в языке программирования C# представляются с помощью ключевых слов for, foreach, while и do, каждое из которых имеет свое специальное представление в CIL. Рассмотрим классический цикл for.

public static void CountToTen() {

 for (int i = 0; i ‹ 10; i++);

}

Вы можете помнить о том, что коды операций br (br, blt и т.д.) используются для управления потоком программы в зависимости от выполнения некоторого условия. В нашем примере мы задали условие, по которому должен произойти выход из цикла, когда значение локальной переменной i станет равным 10. С каждым проходом к значению i добавляется 1, после чего сразу же выполняется тестовое сравнение.

Также напомним, что при использовании любых кодов операций CIL, связанных с ветвлением, нужно определить метку для обозначения в программном коде места, куда следует перейти в случае выполнения условия. С учетом этого рассмотрите следующий (расширенный) программный код CIL, сгенерированный с помощью ildasm.exe (включая и метки программного кода).

.method public hidebysig static void CountToTen() cil managed {

 .maxstack 2

 .locals init ([0] int32 i) // Инициализация локальной целой 'i'.

 IL\_0000: ldc.i4.0 // Загрузка этого значения в стек.

 IL\_0001: stloc.0 // Сохранение значения под индексом '0'.

 IL\_0002: **br.s IL\_0008**// Переход к IL\_0008.

 IL\_0004: ldloc.0 // Загрузка значения с индексом 0.

 IL\_0005: ldc.i4.1 // Загрузка значения '1' в стек.

 IL\_0006: add // Добавление в стек под индексом 0.

 IL\_0007: stloc.0

 IL\_0008: ldloc.0 // Загрузка значения с индексом '0'.

 IL\_0009: ldc.i4.s 10 // Загрузка значения '10' в стек.

 IL\_000b: **blt.s IL\_0004**// Меньше? Если да, то к 1L\_0004.

 IL\_000d: ret

}

В сущности, этот программный код CIL начинается с определения локальной переменной int32 и загрузки ее в стек. Затем осуществляются переходы между командами с метками IL\_0008 и IL\_0004, причем каждый раз значение i увеличивается на 1 и проверяется, осталось ли это значение меньше 10. Если нет, то происходит выход из метода.

**Создание компоновочного блока .NET в CIL**

Теперь, освоив синтаксис и семантику CIL, вы можете закрепить свои знания на практике, построив приложение .NET с использованием только CIL и текстового редактора. Ваше приложение будет состоять из приватного одномодульного \*.dll, содержащего два определения типов класса, и консольного \*.exe, взаимодействующего с этими типами.

**Создание CILCars.dll**

Первым делом следует построить файл \*.dll для использования клиентом. Откройте любой текстовый редактор и создайте новый файл \*.il с именем CILCars.il. Этот одномодульный компоновочный блок будет использовать два внешних двоичных файла .NET, поэтому вы можете начать свой файл программного кода CIL так.

**// Ссылка на mscorlib.dll и**

**// System.Windows.Forms.dll**

.assemblу extern mscorlib {

 .publickeytoken = (B7 7A 5С 56 19 34 E0 89)

 .ver 2:0:0:0

}

.assembly extern System.Windows.Forms {

 .publickeytoken = (B7 7A 5C 56 19 34 E0 89)

 .ver 2:0:0:0

}

// **Определение одномодульного компоновочного блока**.

.assembly CILCars {

 .hash algorithm 0х00008004

 .ver 1:0:0:0

}

.modulе СILCars.dll

Как уже было сказано, этот компоновочный блок будет содержать два типа класса. Первый тип, CILCar, определяет два поля данных и пользовательский конструктор. Второй тип, CarInfoHelper, определяет единственный статический метод с именем DisplayCarInfо(), который использует CILCar в качестве параметра и возвращает void. Оба типа находятся в пространстве имен CILCars. В терминах CIL класс CILCar можно реализовать так.

**// Реализация типа CILCars.CILCar**.

**.**namespace CILCars {

 .class public auto ansi beforefieldinit CILCar extends [mscorlib]System.Object {

**// Поле данных CILCar.**

.field public string petName

  .field public int32 currSpeed

**// Пользовательский конструктор, который дает пользователю**

**// возможность присвоить полю данные.**

  .method public hidebysig specialname rtspecialname instance void .ctor(int32 c, string p) cil managed {

   .maxstack 8

**// Загрузка первого аргумента в стек и вызов**

**// конструктора базового класса.**

   ldarg.0 // объект 'this', а не int32!

   call instance void [mscorlib]System.Object::.ctor()

**// Теперь загрузка первого и второго аргументов в стек.**

   ldarg.0 // объект 'this'

   ldarg.1 // аргумент int32

**// Сохранение элемента вершины стека (int 32) в поле currSpeed.**

   stfld int32 CILCars.CILCar::currSpeed

**// Загрузка строкового аргумента и сохранение в поле petName.**

   ldarg.0 // объект 'this'

   ldarg.2 // аргумент string

   stfld string CILCars.CILCar::petName

   ret

  }

 }

}

Имея в виду, что настоящим первым аргументом любого нестатического члена является объектная ссылка this, в первом блоке CIL-кода мы просто загружаем эту объектную ссылку и вызываем конструктор базового класса. Затем поступающие аргументы конструктора помещаются в стек и запоминаются в полях данных типа с помощью кода операции stfld (сохранение в поле).

Далее, вы должны реализовать второй тип в данном пространстве имен, а именно тип CILCarInfo. Суть типа находится в статическом методе Display(). Основной задачей этого метода является получение поступающего параметра CILCar, извлечение значения поля данных и вывод его в окне сообщения Windows Forms. Вот полная реализация CILCarInfo, а далее следует ее анализ.

.class public auto ansi beforefieldinit CILCarInfo extends [mscorlib]System.Object {

 .method public hidebysig static void Display(class CILCars.CILCar c) cil managed {

  .maxstасk 8

**// Нам нужна локальная строковая переменная.**

  .locals init ([0] string caption)

**// Загрузка строки и входного CILCar в стек.**

  ldstr "Скорость [0]: "

  ldarg.0

**// Помещение значения petName класса CILCar в стек и**

**// вызов статического метода String**.**Format()**.

  ldfld string CILCars.CILCar::petName

  call string [mscorlib]System.String::Format(string, object)

  stloc.0

**// Загрузка значения поля currSpeed и получение его строкового // представления (обратите внимание на вызов ToString()).**

  ldarg.0

  ldflda int32 CILCars.CILCar::currSpeed

  call instance string [mscorlib]System.Int32::ToString()

  ldloc.0

**// Вызов метода MessageBox.Show() с загруженными значениями.**

  call valuetype [System.Windows.Forms] System.Windows.Forms.DialogResult [Sуstem.Windоws.Forms] System.Windows.Forms.MessageBox::Show(string, string)

  pop

  ret

 }

}

Хотя здесь объем программного кода CIL заметно больше, чем в случае реализации CILCar, на самом деле все довольно просто. Во-первых, поскольку вы определяете статический метод, вам не придется иметь дел со скрытой объектной ссылкой (поэтому код операции ldarg.0 действительно загружает поступающий аргумент CILCar).

Метод начинается с загрузки строки ("Скорость {0}: ") в стек за которой следует аргумент CILCar. Когда эти два значения оказываются в нужном месте, загружается значение поля petName и вызывается статический метод System.String. Format(), чтобы вместо замещающих фигурных скобок получить имя CILCar.

Та же общая процедура выполняется и при обработке поля currSpeed, но следует отметить, что здесь используется код. операции ldarga, которая загружает адрес аргумента в стек. Затем вызывается System.Int32.ToString(), чтобы преобразовать значение, размещенное по указанному адресу, в строковый тип. Наконец, когда обе строки отформатированы так, как требуется, вызывается метод MessageBox.Show().

Теперь вы можете скомпилировать свой новый файл \*.dll с помощью ilasm.exe, используя команду

ilasm /dll CILCars.il

а затем проверить полученный CIL-код с помощью peverifу.exe.

peverify CILCars.dll

**Создание CILCarClient.exe**

Теперь нам нужно построить простой компоновочный блок \*.exe, который должен выполнить следующее.

• Создать тип CILCar.

• Передать этот тип статическому методу CILCarInfo.Display(),

Создайте новый файл \*.il и определите внешние ссылки на mscorlib.dll и CILCars.dll (не забудьте поместить копию этого компоновочного блока .NET в каталог приложения клиента!). Затем определите единственный тип (Program), который использует компоновочный блок CILCars.dll. Вот соответствующий программный код, приведенный полностью.

**// Ссылки на внешние компоновочные блоки.**

.assembly extern mscorlib {

 .publickeytoken = (B7 7A 5C 56 19 34 E0 89)

 .ver 2:0:0:0

}

.assembly extern CILCars {

 .ver 1:0:0:0

}

**// Наш выполняемый компоновочный блок.**

.assembly CILCarClient {

 .hash algorithm 0x00008004

 .ver 0:0:0:0

}

.module CILCarClient.exe

**// Реализация типа Program.**

.namespace CILCarClient {

 .class private auto ansi beforefieldinit Program extends [mscorlib]System.Object {

  .method private hidebysig static void Main(string[] args) cil managed {

**// Обозначает точку входа \*.exe.**

   .entrypoint

   .maxstack 8

**// Объявление локального типа CILCar и добавление в стек**

**// значений для вызова конструктора.**

   .locals init ([0] class [CILCars]CILCars.CILCar myCilCar)

   ldc.i4 55

   ldstr "Junior"

**// Создание нового CILCar: сохранение и загрузка ссылки.**

   newobj: instance void [CILCars] CILCars.CILCar::.сtor(int32, string)

   stloc.0

   ldloc.0

**// Вызов Display() и передача верхнего значения из стека.**

   call void [CILCars] CILCars.CILCarInfo::Display(class [CILCars]CILCars.CILCar)

   ret

  }

 }

}

Здесь единственным кодом операции, заслуживающим комментария, является .entrуpoint. Напомним, что этот код операций используется для обозначения метода, который должен выступать в качестве точки входа модуля \*.eхе. Ввиду того, что среда CLR идентифицирует начальный метод для выполнения именно с помощью.entrypoint, сам метод может называться *как угодно*(хотя в нашем примере он называется Main()). В остальном CIL-код метода Main() представляет действия, связанные с добавлением значений в стек и извлечением их из стека.

Заметьте, однако, что для создания CILCar используется код операции.newobj. В связи с этим напомним, что при вызове члена типа непосредственно в CIL вы должны применить синтаксис с использованием двойного двоеточия и, как всегда, указать абсолютное имя типа. Восприняв сказанное, вы можете скомпилировать свой новый файл с помощью ilasm.exe, проверить полученный компоновочный блок с помощью peverifу.exe, а затем выполнить программу.

ilasm CilCarClient.il

peverify CilCarClient.exe

CILCarClient.exe

На рис. 15.5 показан конечный результат.



Рис. 15.5. Ваш CILCar в действии

На этом, выполнив первую задачу этой главы, мы закончим освоение азбуки CIL. К этому моменту, я надеюсь, вы уверенно сможете открыть любой компоновочный блок .NET с помощью ildasm.exe и лучше понимаете, что происходит внутри него.

**Динамические компоновочные блоки**

Как видите, процесс создания сложного приложения .NET непосредственно в CIL оказывается довольно трудоемким. С одной стороны, CIL является чрезвычайно выразительным языком программирования, позволяющим взаимодействовать со всеми программными конструкциями, допустимыми в CTS. С другой стороны, создание CIL-кода является делом скучным, утомительным и сопряженным с множеством ошибок. Хотя верно и то, что знание – это сила, вы можете поинтересоваться, действительно ли это так важно, чтобы "загромождать" законами синтаксиса CIL свою память. Я отвечу так: это зависит от многого. Конечно, для большинства ваших .NET-проектов рассматривать, редактировать или непосредственно создавать программный код CIL не потребуется. Но, освоив азбуку CIL, вы теперь готовы к обсуждению динамических компоновочных блоков (которые называются так в противоположность статическим компоновочным блокам) и роли пространства имен System.Reflection.Emit.

Здесь сразу же может возникнуть вопрос: "В чем разница между статическими и динамическими компоновочными блоками?" По определению, *статические компоновочные блоки*являются двоичными файлами .NET, загружаемыми по запросу CLR непосредственно с диска (в предположении о том, что они размещены где-то на вашем жестком диске в физическом файле или, возможно, во множестве файлов, если компоновочный блок является многомодульным). Как вы можете догадаться сами, каждый раз, когда вы компилируете исходный код C#, вы получаете статический компоновочный блок.

*Динамический компоновочный блок*, с другой стороны, создается в памяти "на лету", с использованием типов, предлагаемых пространством имен System. Reflection.Emit. Пространство имен System.Reflection.Emit делает возможным создание компоновочного блока и его модулей, определений типов и логики реализации CIL прямо в *среде выполнения.*Создав компоновочный блок таким образом, вы можете сохранить свой находящийся в памяти двоичный объект на диск. В результате, конечно, получится новый статический компоновочный блок. Для понимания процесса построения динамического компоновочного блока с помощью пространства имен System.Reflection.Emit требуется определенный уровень понимания кодов операций CIL.

Конечно, создание динамических компоновочных блоков является довольно сложным (и не слишком часто применяемым) приемом программирования, но этот подход может оказаться полезным в следующих обстоятельствах,

• При создании инструментов программирования .NET. позволяющих по требованию динамически генерировать компоновочные блоки в зависимости от пользовательского ввода.

• При создании программ, способных динамически генерировать агенты доступа к удаленным типам на основе получаемых метаданных.

• При загрузке статических компоновочных блоков с динамическим добавлением новых типов в двоичный образ.

Учитывая сказанное, давайте рассмотрим типы, предлагаемые в System.Reflection.Emit.

**Исследование пространства имен System.Reflection.Emit**

Для создания динамического компоновочного блока требуется в определенной мере понимать коды операций CIL, но типы пространства имен System.Reflection. Emit в максимальной мере "пытаются" скрыть сложность CIL. Например, вместо прямого указания необходимых директив и атрибутов CIL при определении типа класса вы можете просто использовать класс TypeBuilder. Точно так же, чтобы определить новый конструктор уровня экземпляра, нет никакой необходимости использовать specialname, rtspecialname и лексемы .ctor – вместо этого можно просто использовать ConstructorBuilder. Описания ключевых членов пространства имен System.Reflection.Emit приводятся в табл. 15.8.

Таблица 15.8. Избранные члены пространства имен System.Reflection.Emit

|  |  |
| --- | --- |
| **Члены** | **Описание** |
| AssemblyBuilder | Используется для создания компоновочного блока (\*.dll или \*.exe) в среде выполнения. В случае \*.exe следует вызвать метод ModuleBuilder.SetEntryPoint(), чтобы указать метод, являющийся точкой входа в модуль. Если точка входа не указана, будет сгенерирована \*.dll |
| ModuleBuilder | Используется для определения множества модулей в рамках данного компоновочного блока |
| EnumBuilder | Используется для создания типа перечня .NET |
| TypeBuilder | Может использоваться дли создания классов, интерфейсов, структур и делегатов в рамках модуля в среде выполнения |
| MethodBuilder EventBuilder LocalBuilder PropertyBuilder FieldBuilder ConstructorBuilder CustomAttributeBuilder ParameterBuilder | Используются для создания членов типа (таких как методы, локальные переменные, свойства, конструкторы и атрибуты) в среде выполнения |
| ILGenerator | Генерирует коды операций CIL в данном члене типа |
| OpCodes | Обеспечивает множество полей, отображающихся в коды операций CIL. Этот тип используется вместе с различными членами System.Reflection.Emit.ILGenerator |

В общем, типы пространства имен System.Reflection.Emit при построении динамического двоичного модуля позволяют представлять "сырые" лексемы CIL программными единицами. Возможности использования многих из указанных членов будут продемонстрированы в следующем примере, но тип ILGenerator заслуживает отдельного обсуждения.

**Роль System.Reflection.Emit.ILGenerator**

Как следует из самого имени указанного типа, роль ILGenerator заключается в добавлении кодов операций CIL в данный член типа. Обычно нет необходимости непосредственно создавать объект ILGenerator, а нужно просто получить действительную ссылку на тип ILGenerator, используя типы, связанные с компоновщиком (такие как MethodBuilder и ConstructorBuilder). Например:

**// Получение ILGenerator из объекта ConstructorBuilder**

**// с именем 'myCtorBuilder'.**

ConstructorBuilder myCtorBuilder = new ConstructorBuilder (/\*…различные аргументы… \*/);

 ILGenerator myCILGen = myCtorBuilder.**GetILGenerator();**

Имея ILGenerator, вы можете генерировать "сырые" коды операций CIL, используя любые из целого набора методов. Некоторые (но, конечно же, не все) методы ILGenerator описаны в табл. 15.9.

**Таблица 15.9.**Подборка методов ILGenerator

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| BeginCatchBlock() | Начинает блок catch |
| BeginExceptionBlock() | Начинает блок неотфильтрованного исключения |
| BeginFinallyBlock() | Начинает блок finally |
| BeginScope() | Начинает лексический контекст |
| DeclareLocal() | Объявляет локальную переменную |
| DefineLabel() | Объявляет новую метку |
| Emit() | Является перегруженным и позволяет генерировать коды операций CIL |
| EmitCall() | Вставляет код операции call или callvirt в поток CIL |
| EmitWriteLine() | Генерирует вызов Console.WriteLine() с различными типами значений |
| EndExceptionBlock() | Завершает блок исключения |
| EndScope() | Завершает лексический контекст |
| ThrowException() | Создает инструкцию для генерирования исключения |
| UsingNamespace() | Указывает пространство имен, которое будет использоваться для оценки локальных переменных и наблюдаемых значений в текущем активном лексическом контексте |

Ключевым методом ILGenerator является метод Emit(), который работает в совокупности с типом класса System.Reflection.Emit.OpCodes. Как уже упоминалось в этой главе, данный тип открывает большой набор доступных только для чтения полей, отображающихся в коды операций CIL. Полностью эти члены описаны в оперативно доступной системе справки, но целый ряд примеров вы сможете увидеть и на следующих страницах.

**Генерирование динамического компоновочного блока**

Чтобы проиллюстрировать процесс определения компоновочного блока .NET в среде выполнения, давайте создадим одномодульный динамический компоновочный блок с именем MyAssembly.dll. В этом модуле будет содержаться класс HelloWorld. Тип HelloWorld поддерживает конструктор, используемый по умолчанию, и пользовательский конструктор для присваивания значения приватной переменной (theMessage) типа string. Кроме того, HelloWorld предлагает открытый метод экземпляра с именем SayHello(), который выводит приветствие в стандартный поток ввода-вывода, а также еще один метод экземпляра, GetMsg(), который возвращает внутреннюю приватную строку. В результате вы должны программно сгенерировать следующий тип класса.

**// Этот класс будет создан в среде выполнения**

**// с помощью System.Reflection.Emit.**

public class HelloWorld {

 private string theMessage;

 HelloWorld() {}

 HelloWorld(string s) { theMessage = s; }

 public string GetMsg() { return theMessage; }

 public void SayHello() {

  System.Console.WriteLine("Привет от класса HelloWorld!");

 }

}

Предположим, вы cоздали новый проект консольного приложения в Visual Studio 2005, назвав его DynAsmBuilder. Переименуйте исходный класс в MyAsmBuilder и определите статический метод с именем CreateMyAsm(). Этот единственный метод будет ответственен за следующее:

• определение характеристик динамического компоновочного блока (имя, версия и т.д.);

• реализацию тина HelloClass;

• запись компоновочного блока, сгенерированного в памяти, в физический файл.

Также отметим, что метод CreateMyAsm() использует в качестве единственного параметра тип System.AppDomain, который будет использоваться для получения доступа к типу AssemblyBuilder, связанному с текущим доменом приложения (см. главу 13, где обсуждаются домены приложений .NET). Вот полный программный код, с последующим анализом.

**// Вызывающая сторона посылает тип AppDomain.**

public static void CreateMyAsm(AppDomain currAppDomain) {

**// Установка общих характеристик компоновочного блока.**

 AssemblyName assemblyName = new AssemblyName();

 assemblyName.Name = "MyAssembly";

 assemblyName.Version = new Version("1.0.0.0");

**// Создание нового компоновочного блока**

**// в рамках текущего домена приложения.**

 AssemblyBuilder assembly = curAppDomain.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Save);

**// Поскольку создается одномодульный компоновочный блок,**

**// имя модуля будет совпадать с именем компоновочного блока.**

 ModuleBuilder module = assembly.DefineDynamicModule("MyAssembly", "MyAssemblу.dll");

**// Определение открытого класса с именем "HelloWorld".**

 TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld", TypeAttributes.Public);

**// Определение приватной переменной String с именем "theMessage"**.

 FieldBuilder msgField = helloWorldClass.DefineField("theMessage", Type.GetType("System.String"), FieldAttributes.Private);

**// Создание пользовательского конструктора.**

 Type[] constructorArgs = new Type[1];

 constructorArgs[0] = typeof(string);

 ConstructorBuilder constructor = helloWorldClass.DefineConstructor(MethodAttributes.Public, CallingConventions.Standard, constructorArgs);

 ILGenerator constructorIL = constructor.GetILGenerator();

 constructorIL.Emit(OpCodes.Ldarg\_0);

 Type objectClass = typeof(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);

**// Создание конструктора, заданного по умолчанию.**

 helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);

**// Теперь создание метода GetMsg().**

 MethodBuilder getMsgMethod = helloWorldClass.DefineMethod("GetMsg", MethodAttributes.Public, typeof(string), null);

 ILGenerator methodIL = getMsgMethod.GetILGenerator();

 methodIL.Emit(OpCodes.Ldarg\_0);

 methodIL.Emit(OpCodes.Ldfld, msgField);

 methodIL.Emit(Opcodes.Ret);

**// Создание метода SayHello.**

 MethodBuilder sayHiMethod = helloWorldClass.DefineMethod("SayHello", MethodAttributes.Public, null, null);

 methodIL = sayHiMethod.GetILGenerator();

 methodIL.EmitWriteLine("Привет от класса HelloWorld!");

 methodIL.Emit(Opcodes.Ret);

**// Генерирование класса HelloWorld.**

 helloWorldClass.CreateType();

**// (Необязательно.) Сохранение компоновочного блока в файл.**

 assembly.Save("MyAssembly.dll");

}

**Генерирование компоновочного блока и набора модулей**

Метод начинается с указания минимального набора характеристик компоновочного блока, для чего используются типы AssemblyName и Version (определенные в пространстве имен System.Reflection). Затем с помощью метода уровня экземпляра AppDomain.DеfineDynamicAssembly() вы получаете тип AssemblyBuilder (напомним, что вызывающая сторона передаст в метод CreateMyAsm() ссылку на AppDomain).

**// Установка общих характеристик компоновочного блока**

**// и получение доступа к типу AssemblyBuilder.**

public static void CreateMyAsm(AppDomain **currAppDomain**) {

 AssemblyName assemblyName = new AssemblyName();

 assemblyName.Name = "MyAssembly";

 assemblyName.Version = new Version("1.0.0.0");

**// Создание нового компоновочного блока в текущем AppDomain.**

 AssemblyBuilder assembly = **currAppDomain**.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Save);

 …

}

Как видите, при вызове AppDomain.DefineDynamicAssembly() вы должны указать режим доступа к компоновочному блоку. Этот режим может задаваться любым из значений, указанных в табл. 15.10.

**Таблица 15.10.**Значения перечня AssemblyBuilderAccess

|  |  |
| --- | --- |
| **Значение** | **Описание** |
| ReflectionOnly | Динамический компоновочный блок может только отображаться |
| Run | Динамический компоновочный блок может выполняться в памяти, но не сохраняться на диск |
| RunAndSave | Динамический компоновочный блок может выполняться в памяти и сохраниться на диск |
| Save | Динамический компоновочный блок может сохраняться на диск, но не выполняться в памяти |

Следующей задачей является определение набора модулей для нового компоновочного блока. Поскольку данный компоновочный блок является одномодульным, вы должны определить только один модуль. Если с помощью метода DefineDynamicModule() требуется построить многомодульный компоновочный блок, вы должны указать необязательный второй параметр, задающий имя данного модуля (например, myMod.dotnetmodule). Однако при создании одномодульного компоновочного блока имя модуля будет идентично имени самого компоновочного блока. Так или иначе, после завершения работы метода DefineDynamicModule() вы получите ссылку на действительный тип ModuleBuilder.

**// Одномодульный компоновочный блок.**

ModuleBuilder module = assembly **.DefineDynamicModule**("MyAssembly", "MyAssembly.dll");

**Роль типа ModuleBuilder**

Тип ModuleBuilder является ключевым типом для процесса построения динамических компоновочных блоков. В соответствии с возможными ожиданиями, ModuleBuilder предлагает целый ряд членов, позволяющих определить множество типов, содержащихся в данном модуле (классы, интерфейсы, структуры и т.д.), а также множество встроенных ресурсов (таблицы строк, изображения и т.д.; формат ресурсов .NET будет рассмотрен в главе 20). Некоторые из методов, относящихся к созданию инфраструктуры модуля, описаны в табл. 15.11 (каждый из этих методов возвращает тип, представляющий тот тип, который вы собирались сконструировать).

**Таблица 15.11**. Подборка членов типа ModuleBuilder

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| DefineEnum() | Используется для генерирования определения перечня .NET |
| DefineResource() | Определяет управляемый встроенный ресурс, который должен храниться в данном модуле |
| DefineType() | Конструирует TypeBuilder, который позволяет определять типы значений, интерфейсы и типы класса (в том числе и делегаты) |

Ключевым членом класса ModuleBuilder, о котором следует знать, является DefineType(). Вдобавок к указанию имени типа (в виде простой строки), вы должны использовать перечень System.Reflection.TypeAttributes, чтобы непосредственно описать формат типа. Основные члены перечня TypeAttributes представлены в табл. 15.12.

**Таблица 15.12.**Подборка элементов перечня TypeAttributes

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Abstract | Указывает абстрактный тип |
| Class | Указывает тип класса |
| Interface | Указывает тип интерфейса |
| NestedAssembly | Указывает, что класс вложен в область видимости компоновочного блока и поэтому доступен только для методов соответствующего компоновочного блока |
| NestedFamAndAssem | Указывает, что класс вложен в область видимости семейства и компоновочного блока и поэтому доступен только для методов, принадлежащих пересечению соответствующего семейства и компоновочного блока |
| NestedFamily | Указывает, что класс вложен в область видимости семейства и поэтому доступен только для методов соответствующего типа и его подтипов |
| NestedFamORAssem | Указывает, что класс вложен в область видимости семейства или компоновочного блока и поэтому доступен только для методов, принадлежащих объединению соответствующего семейства и компоновочного блока |
| NestedPrivate | Указывает вложенный класс с приватной областью видимости |
| NestedPublic | Указывает вложенный класс с общедоступной областью видимости |
| NotPublic | Указывает класс, не являющийся открытым |
| Public | Указывает открытый класс |
| Sealed | Указывает изолированный класс, который не может быть расширен |
| Serializable | Указывает класс, допускающий сериализацию |

**Генерирование типа HelloClass и принадлежащей ему строковой переменной**

Теперь вы понимаете роль метода ModuleBuilder.CreateType(), и пришло время выяснить, как сгенерировать открытый тип класса HelloWorld и приватную строковую переменную.

**// Определение открытого класса MyAssembly.HelloWorld.**

TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld", TypeAttributes.Public);

**// Определение принадлежащей классу приватной переменной String**

**// с именем theMessage.**

FieldBuilder msgField =hellоWоrldclass.DefineField("theMessage", typeof(string), FieldAttributes.Private);

Обратите внимание на то, что метод TypeBuilder.DefineField() обеспечивает доступ к типу FieldBuilder. Класс TypeBuilder определяет также другие методы, обеспечивающие доступ к другим типам "построителя". Например, DefineConstructor() возвращает ConstructorBuilder.DefineProperty() – PropertyBuilder и т.д.

**Генерирование конструкторов**

Как уже упоминалось выше, для определения конструктора типа может использоваться метод TypeBuilder.DefineConstructor(). Однако в нашей реализации конструктора HelloClass, чтобы назначить поступающий параметр внутренней приватной строке, мы добавим CIL-код в тело конструктора непосредственно. Чтобы получить тип ILGenerator, вызывается метод GetILGenerator() соответствующего типа "построителя", на который имеется ссылка (в данном случае это тип ConstructorBuilder).

Метод Emit() класса ILGenerator отвечает за размещение CIL-кода в реализации члена. Сам метод Emit() часто использует тип класса OpCodes, который с помощью полей, доступных только для чтения, открывает доступ к набору кодов операций CIL. Например, OpCodes.Ret указывает возврат вызова метода, OpCodes.Stfld выполняет присваивание значения члену-переменной, a OpCodes.Call используется для вызова метода (в нашем случае это конструктор базового класса). С учетом сказанного рассмотрите следующую программную логику конструктора.

**// Создание пользовательского конструктора, имеющего**

**// один аргумент System.String.**

Type[] constructorArgs = new Type[1];

constructorArgs[0] *=*typeof(string);

ConstructorBuilder constructor = helloWorldClass.DefineConstructor(MethodAttributes.Public, CallingConventions.Standard, constructorArgs);

**// Теперь в конструктор добавляется необходимый CIL-код.**

ILGenerator constructorIL = constructor.GеtILGenerator();

constructorIL.Emit(**OpCodes.Ldarg\_0**);

Type objectClass = typeof(object);

ConstructorInfo superConstructor = objectClass.GetConstructor(new Type[0]);

constructorIL.Emit(**OpCodes.Call**, superConstructor);**// Вызов конструктора базового класса.**

**// Загрузка указателя 'this' объекта в стек.**

constructorIL.Emit**(OpCodes.Ldarg\_0);**

**// Загрузка входного аргументе в стек и сохранения в msgField.**

constructorIL.Emit(Opcodes.Ldarg\_1);

constructorIL.Emit(Opcodes.Stfld, msgField);**// Присвоение msgField.**

constructorIL.Emit(Opcodes.Ret);**// Возврат.**

Вы, конечно, хорошо знаете, что как только для типа определяется пользовательский конструктор, конструктор, заданный по умолчанию, автоматически "отключается". Чтобы переопределить конструктор, не имеющий аргументов, просто вызовите метод DefineDefaultConstructor() типа TypeBuilder, как показано ниже.

**// Восстановление конструктора, заданного по умолчанию.**

helloWorldClass.**DefineDefaultConstructor**(MethodAttributes.Public);

Следующий вызов порождает стандартный CIL-код для определения конструктора, заданного по умолчанию.

.method public hidebysig specialname rtspecialname instance void .ctor() cil managed {

 .maxstack 1

 ldarg.0

 call instance void [mscorlib]System.Object::.ctor()

 ret

}

**Генерирование метода HelloWorld()**

Наконец, рассмотрим задачу генерирования метода SayHello(). Первой задачей здесь оказывается получение типа MethodBuilder из переменной helloWorld-Class. После этого определяется указанный метод и получается ILGenerator, позволяющий добавить соответствующие CIL-инструкции.

**// Создание метода SayHello.**

MethodBuilder sayHiMethod = helloWorldClass.DefineMethod("SayHello", MethodAttributes.Public, null, null);

methodIL = sayHiMethod.GetILGenerator();

**// Вывод на консоль.**

methodIL.EmitWriteLine("Всем привет!");

methodIL.Emit(Opcodes.Ret);

Здесь создается открытый метод (MethodAttributes.Public), не имеющий параметров и не возвращающий ничего (на это указывают значения null в вызове DefineMethod()). Также обратите внимание на вызов EmitWriteLine()*.*Этот вспомогательный член класса ILGenerator автоматически записывает строку в стандартный поток вывода.

**Использование динамически сгенерированного компоновочного блока**

Теперь, когда имеется вся программная логика, позволяющая создать и сохранить компоновочный блок, нужен класс для запуска этой логики. Для того чтобы замкнуть цикл, предположим, что в проекте определен второй класс, названный AsmReader. С помощью метода Thread.GetDomain() в Main() получается доступ к текущему домену приложения, который используется для принятия динамически создаваемого компоновочного блока. Получив соответствующую ссылку, вы можете вызвать метод CreateMyAsm().

Чтобы сделать процесс немного более интересным, после завершения вызова CreateMyAsm() будет выполнено динамическое связывание (см. главу 12), обеспечивающее загрузку нового компоновочного блока в память и взаимодействие с членами класса HelloWorld.

using System;

using System.Reflection.Emit;

using System.Reflection;

using System*.*Threading;

…

public class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\*\*\*\*\* Чудесный построитель \*\*\*\*\*\*\*\*\*\*");

  Console.WriteLine ("\*\*\* динамических компоновочных блоков \*\*\*");

**// Получение домена приложения для данного потока.**

  AppDomain currAppDomain = Thread.GetDomain();

**// Создание динамического компоновочного блока с помощью f(х).**

  СreateMyAsm(currAppDomain);

  Console.WriteLine("-› Завершение создания MyAssembly.dll.");

**// Теперь загрузка нового компоновочного блока из файла.**

  Console.WriteLine("-› Загрузка MyAssembly.dll из файла.");

  Assembly a = Assembly.Load("MyAssembly");

**// Получение типа HellоWorld.**

  Type hello = a.GetType("MyAssembly.HelloWorld");

**// Создание объекта HelloWorld и вызов нужного конструктора**.

  Console.Write("-› Введите сообщение для класса HelloWorld: ");

  string msg *=*Console.ReadLine();

  object[] ctorArgs = new object[1];

  ctorArgs[0] = msg;

  object obj = Activator.CreateInstance(hello, ctorArgs);

**// Вызов SayHello и вывод возвращенной строки.**

  Console.WriteLine("-› Вызов SayHello()");

  Console.WriteLine(" через динамическое связывание.");

  MethodInfo mi = hello.GetMethod("SayHello");

  mi.Invoke(obj, null);

**// Подключение GetMsg(). Метод Invoke() возвращает объект,**

**// содержащий возвращенное значение метода.**

  mi = hello.GetMethod("GetMsg");

  Console.WriteLine(mi.Invoke(obj, null));

 }

}

В результате создается компоновочный блок .NET, способный создавать компоновочные блоки .NET в среде выполнения.

На этом наш обзор CIL и роли динамических компоновочных блоков завершается. Я надеюсь, эта глава позволила расширить горизонты вашего понимания системы типов .NET, а также синтаксиса и семантики CIL.

**Замечание.**Обязательно загрузите свой динамически созданный компоновочный блок в ildasm.exe, чтобы выяснить, как функциональные возможности пространства имен System. Reflection.Emit реализуются в программном коде CIL

**Исходный код.**Проект DynAsmBuilder размещен в подкаталоге, соответствующем главе 15.

**Несколько слов о System.CodeDOM**

Теперь, когда мы с вами выяснили, как создаются динамические компоновочные блоки с помощью System.Reflection.Emit и различных лексем CIL, я должен сообщить вам, что есть и другая (часто более простая) альтернатива. Платформа .NET предлагает технологию под названием *модель DOM для программного кода*(модель *code DOM),*которая позволяет представить структуру .NET-типа в независимых от языка терминах с помощью объектного графа. Построив такой граф с помощью членов пространства имен System.CodeDOM, вы получаете возможность динамически перевести его содержимое в файл программного кода, соответствующего любому языку (C#, Visual Basic .NET или любому языку стороннего поставщика, обеспечившего поддержку *code DOM*)*.*Кроме того, пространство имен System.CodeDOM.Compiler и связанные с ним другие пространства имен могут использоваться для компиляции объектного графа, находящегося в памяти (или сохраненного) объекта в действительный статический компоновочный блок .NET.

К сожалению, в этой книге нет места для подробного обсуждения технологии *code DOM.*Поэтому если вам нужна дополнительная информация, выполните поиск по ключу "CodeDOM, quick reference" в документации .NET Framework 2.0 SDK.

**Резюме**

В этой главе предлагается краткий обзор возможностей синтаксиса и семантики CIL. В отличие от управляемых языков высшего уровня, таких как, например, C#, в CIL не просто определяется набор ключевых слов, но и директивы (для определения структуры компоновочного блока и его типов), атрибуты (уточняющие характеристики соответствующей директивы) и коды операций (используемые для реализации членов типов). Был также рассмотрен компилятор CIL (ilasm.exe). Вы узнали о том, как изменить содержимое компоновочного блока .NET, непосредственно изменяя его программный код CIL, и рассмотрели основные этапы, процесса построения компоновочного блока .NET с помощью CIL.

Вторая половина главы была посвящена обсуждению пространства имен System.Reflection.Emit. Используя соответствующие типы, вы можете создавать компоновочные блоки .NET в памяти динамически. При желании можно также сохранить созданный в памяти образ в физическом файле на диске. Многие типы System.Reflection.Emit автоматически генерируют подходящие директивы и атрибуты CIL, используя другие связанные с ними типы, такие как ConstructorBuilder, TypeBuilder и т.д. Тип ILGenerator может использоваться для добавления необходимых кодов операций CIL в члены типа. И хотя существует целый ряд вспомогательных типов, призванных упростить процесс создания программ при использовании кодов операций CIL, для успешного создания динамических компоновочных блоков вам понадобится хорошее понимание языка CIL.

**ЧАСТЬ IV. Программирование с помощью библиотек .NET**

**ГЛАВА 16. Пространство имен System.IO**

При создании полноценных приложений исключительно важна возможность сохранения информации между сеансами доступа пользователя. В этой главе рассматривается целый ряд вопросов, связанных с реализацией ввода-вывода в .NET. Первой нашей задачей будет исследование базовых типов, определенных в пространстве имен System.IO, и выяснение того, как программными средствами можно изменить рабочий каталог и структуру файлов машины. После этого мы рассмотрим различные возможности чтения и записи данных из файлов с сим-вольной, двоичной и строковой организацией, а также из памяти.

**Анализ пространства имен System.IO**

В .NET пространство имен System.IO является той частью библиотек базовых адресов, которая обслуживает службы ввода-вывода, как для файлов, так и для памяти. Подобно любому другому пространству имен, System.IO определяет свой набор классов, интерфейсов, перечней, структур и делегатов, большинство из которых содержится в mscorlib.dll. Вдобавок к типам, содержащимся в mscorlib.dll, часть членов System.IO содержится в компоновочном блоке System.dll (все проекты в Visual Studio 2005 автоматически устанавливают ссылку на оба эти компоновочных блока, поэтому вам об этом беспокоиться не приходится).

Задачей многих типов, принадлежащих System.IO), является программная поддержка физических операций с каталогами и файлами. Но есть и другие типы, обеспечивающие поддержку операций чтения и записи данных строковых буферов, a также непосредственный доступ к памяти. Чтобы представить вам общую картину функциональных возможностей пространства имен System.IO, в табл. 16.1 описаны его базовые (неабстрактные) классы.

Вдобавок к этим типам, допускающим создание экземпляров, в System.IO определяется целый ряд перечней, а также набор абстрактных классов (Stream, TextReader, TextWriter и т.д.), которые обеспечивают открытый полиморфный интерфейс всем своим производным классам. Более подробная информация об этих типах будет предлагаться в процессе дальнейшего обсуждения материала этой главы.

**Таблица 16.1.**Ключевые типы пространства имен System.IO

|  |  |
| --- | --- |
| **Неабстрактный тип класса ввода-вывода** | **Описание** |
| BinaryReader BinaryWriter | Позволяют сохранять и читать примитивные типы данных (целые, логические, строковые и другие), как двоичные значения |
| BufferedStream | Обеспечивает временное хранилище для потока байтов, которые можно будет направить в другое хранилище позже |
| Directory DirectoryInfо | Используются для работы со структурой каталогов машины. Тип Directory предлагает свои функциональные возможности, в основном через *статические методы.*Тип DirectoryInfo обеспечивает аналогичные возможности с помощью подходящей *объектной переменной* |
| DriveInfo | Этот тип (появившийся в .NET 2.0) предлагает подробную информацию о дисках, установленных на машине |
| File FileInfo | Используются для работы с файлами. Тип File предлагает свои функциональные возможности, в основном через *статические методы.*Тип FileInfo обеспечивает аналогичные возможности с помощью подходящей *объектной переменной* |
| FileStream | Позволяет реализовать произвольный доступ к файлам (например, поиск), когда данные представлены в виде потока байтов |
| FileSystemWatcher | Позволяет контролировать изменения внешнего файла |
| MemoryStream | Обеспечивает прямой доступ к данным, сохраненным в памяти, а не в физическом файле |
| Path | Выполняет операции с типами System.String, содержащими информацию о файлах или каталогах в независимом от платформы виде |
| StreamWriter StreamReader | Используются для записи (и чтения) текстовой информации файлов. Эти типы не поддерживают доступ к файлам с произвольной организацией |
| StringWriter StringReader | Подобно типам StreamReader/StreamWriter, эти классы тоже обеспечивают обработку текстовой информации. Однако соответствующим хранилищем в данном случае является строковый буфер, а не физический файл |

**Типы Directory(Info) и File(Info)**

Пространство System.IO предлагает четыре типа, позволяющие как обработку отдельных файлов, так и взаимодействие со структурой каталогов машины. Первые два из этих типов – Directory и File – с помощью различных статических членов позволяют выполнение операций создания, удаления, копирования и перемещения файлов. Родственные типы FileInfo и DirectoryInfo предлагают аналогичные возможности в виде методов экземпляра (который, таким образом, необходимо будет создать). На рис. 16.1 показана схема зависимости типов, связанных с обработкой каталогов и файлов. Обратите внимание на то, что типы Directory и File расширяют непосредственно System.Object, в то время как DirectoryInfo и FileInfo получаются из абстрактного типа FileSystemInfo.

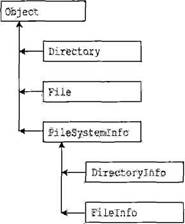


Рис. 16.1. Типы, обеспечивающие работу с каталогами и файлами

Вообще говоря, FileInfо и DirectoryInfо являются лучшим выбором для рекурсивных операций (таких как, например, составление перечня всех подкаталогов с данным корнем), поскольку члены классов Directory и File обычно возвращает строковые значения, а не строго типизированные объекты.

**Абстрактный базовый класс FileSystemInfo**

Типы DirectoryInfo и FileInfo во многом наследуют свое поведение от абстрактного базового класса FileSystemInfo. По большей части члены класса FileSystemInfo используются для получения общих характеристик (таких как, например, время создания, различные атрибуты и т.д.) соответствующего файла иди каталога. В табл. 16.2 описаны свойства FileSystemInfo, представляющие наибольший интерес.

**Таблица 16.2.**Свойства FileSystemInfo

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Attributes | Читает или устанавливает атрибуты, связанные с текущим файлом, представленным в перечне FileAttributes |
| CreationTime | Читает или устанавливает время создания для текущего файла или каталога |
| Exists | Может использоваться для выяснения того, существует ли данный файл или каталог |
| Extension | Читает расширение файла |
| FullName | Получает полный путь каталога или файла |
| LastAccesTime | Читает или устанавливает время последнего доступа к текущему файлу или каталогу |
| LastWriteTime | Читает или устанавливает время последнего сеанса записи в текущий файл или каталог |
| Name | Для файлов получает имя файла. Для каталогов получает имя последнего каталога в иерархии, если такая иерархия существует. Иначе получает имя каталога |

Тип FileSystemInfo определяет также метод Delete(). Этот метод реализуется производными типами для удаления данного файла или каталога с жесткого диска. Кроме того, перед получением информации атрибута может вызываться Refresh(), чтобы гарантировать то, что информация о текущем файле (или каталоге) не будет устаревшей.

**Работа с типом DirectoryInfo**

Первым из рассматриваемых в нашем обсуждении типов, связанных с реализацией ввода-вывода и допускающих создание экземпляров, будет класс DirectoryInfo. Этот класс предлагает набор членов, используемых для создания, перемещения, удаления и перечисления каталогов и подкаталогов. Кроме функциональных возможностей, обеспеченных базовым классом (FileSystemInfo), класс DirectoryInfo предлагает и свои члены, описанные в табл. 16.3.

**Таблица 16.3.**Основные члены типа DirectoryInfo

|  |  |
| --- | --- |
| **Члены** | **Описание** |
| Create() CreateSubdirectory() | Создает каталог (или множество подкаталогов) в соответствии с заданным именем пути |
| Delete() | Удаляет каталог и все его содержимое |
| GetDirectories() | Возвращает массив строк, представляющих все подкаталоги текущего каталога |
| GetFiles() | Получает массив типов FileInfo, представляющих множество файлов данного каталога |
| MoveTo() | Перемещает каталог и его содержимое в место, соответствующее заданному новому пути |
| Parent | Получает каталог родителя указанного пути |
| Root | Получает корневую часть пути |

Мы начнем работу с типом DirectoryInfo с попытки указать конкретный путь каталога для использования в качестве параметра конструктора. Чтобы получить доступ к текущему каталогу приложения (т.е. к каталогу приложения, выполняющегося в данный момент), используйте обозначение ".". Вот подходящие примеры.

**// Привязка к текущему каталогу приложения.**

DirectoryInfo dir1 = new DirectoryInfo(".");

**// Привязка к C:\Windows с помощью строки,**

**// для которой указано "дословное" применение.**

DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");

Во втором примере предполагается, что передаваемый конструктору путь (путь C:\Windows) уже существует на данной физической машине. Если вы попытаетесь взаимодействовать с несуществующим каталогом, будет сгенерировано исключение System.IO.DirectoryNotFoundException (каталог не найден). Поэтому если вы укажете каталог, который еще не создан, то перед его использованием вам придется сначала вызвать метод Create().

**// Привязка к несуществующему каталогу с последующим его созданием.**

DirectoryInfo dir3 = new DirectoryInfo(@"C:\Window\Testing");

dir3.Create();

После создания объекта DirectoryInfo вы можете исследовать содержимое соответствующего каталога с помощью свойств, унаследованных от FileSystemInfo. Например, следующий класс создает новый объект DirectoryInfo, связанный с C:\Windows (при необходимости измените этот путь в соответствии с установками системы на вашей машине) и отображающий ряд интересных статистических данных об указанном каталоге (рис. 16.2).

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\**\*\*\*\**Забавы с Directory(Info) \*\*\*\*\*\n");

  DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");

**// Информация о каталоге.**

  Console.WriteLine("\*\*\*\*\* Информация о каталоге \*\*\*\*\*");

  Console.WriteLine("Полное имя: {0} ", dir.FullName);

  Console.WriteLine("Имя: {0} ", dir.Name);

  Console.WriteLine("Родитель: {0} ", dir.Parent);

  Console.WriteLine("Создан: {0} "*,*dir.CreationTime);

  Console.WriteLine("Атрибуты: {0} ", dir.Attributes);

  Console.WriteLine("Корневой каталог: {0}", dir.Root);

  Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

 }

}



Рис. 16.2. Информация о каталоге Windows

**Перечень FileAttributes**

Свойство Attributes, предоставленное объектом FileSystemInfо, обеспечивает получение различной информации о текущем каталоге или файле, и вся она содержится в перечне FileAttributes. Имена полей этого перечня говорят сами за себя, но некоторые менее очевидные имена здесь сопровождаются комментариями (подробности вы найдете в документации .NET Framework 2.0 SDK).

public enum FileAttributes {

 ReadOnly,

 Hidden,

**// Файл, являющийся частью операционной системы или используемый**

**// исключительно операционной системой.**

 System,

 Directory,

 Archive,

**// Это имя зарезервировано для использования в будущем.**

 Device,

**// Файл является 'нормальным' (если не имеет других**

**// установленных атрибутов),**

 Normal,

 Temporary,

**// Разреженные файлы обычно являются большими файлами,**

**// данные которых по большей части – нули.**

 SparseFile,

**// Блок пользовательских данных, связанных с файлом или каталогом.**

 ReparsePoint,

 Compressed,

 Offline,

**// Файл, который не будет индексирован службой индексации**

**// содержимого операционной системы.**

 NotContentIndexed,

 Encrypted

}

**Перечисление файлов с помощью DirectoryInfo**

Вдобавок к получению базовой информации о существующем каталоге, вы можете добавить в пример несколько вызовов методов типа DirectoryInfo. Сначала используем метод GetFiles(), чтобы получить информацию обо всех файлах \*.bmp, размещенных каталоге C:\Windows. Этот метод возвращает массив типов FileInfo, каждый из которых сообщает подробности о конкретном файле (подробности о самом типе FileInfo будут представлены в этой главе немного позже).

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Забавы с Directory(Info) \*\*\*\*\*\n");

  DirectoryInfo dir = new DireetoryInfо(@"C:\Windows");

**// Получение всех файлов с расширением bmp.**

  FileInfo[] bitmapFiles = dir.GetFiles("\*.bmp");

**// Сколько их всего?**

  Console.WriteLine("Найдено {0} файлов \*.bmp\n", bitmapFiles.Length);

**// Вывод информации о файлах.**

  foreach (FileInfo f in bitmapFiles) {

   Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

   Console.WriteLine("Имя: {0} ", f.Name);

   Console.WriteLine("Размер: {0} ", f.Length);

   Console.WriteLine("Создан: {0} ", f.CreationTime);

   Console.WriteLine("Атрибуты: {0} ", f.Attributes);

   Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

  }

 }

}

Запустив это приложение, вы увидите список, подобный показанному на рис. 16.3 (ваши результаты могут быть другими!).



Рис. 16.3. Информация о файлах с точечными изображениями

**Создание подкаталогов с помощью DirectoryInfo**

Вы можете программно расширить структуру каталога, используя метод DirectoryInfo.CreateSubdirectory(). Этот метод с помощью одного обращения к функции позволяет создать как один подкаталог, так и множество вложенных подкаталогов. Для примера рассмотрите следующий блок программного кода, расширяющий структуру каталога C:\Windows путем создания нескольких пользовательских подкаталогов.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Забавы с Directory(Info) \*\*\*\*\*\n");

  DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");

  …

**// Создание \MyFoo в исходном каталоге.**

  dir.CreateSubdirectory("MyFoo");

**// Создание \MyBar\MyQaaz в исходном каталоге**

  dir.CreateSubdirectory(@"MyBar\MyQaaz");

 }

}

Если теперь проверить каталог Windows в окне программы Проводник, вы увидите там новые подкаталоги (рис. 16.4).

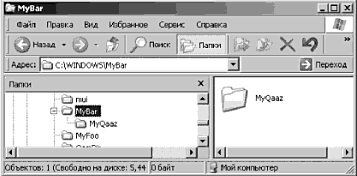


Рис. 16.4. Создание подкаталогов

Хотя вы и не обязаны использовать возвращаемое значение метода CreateSubdirectory(), полезно знать, что в случае успешного выполнения тип DirectoryInfo возвращает созданный элемент.

**// CreateSubdirectory() возвращает объект DirectoryInfo,**

**// представляющий новый элемент.**

DirectoryInfo d = dir.CreateSubdirectory("MyFoo");

Console.WriteLine("Создан: {0} ", d.FullName);

d = dir.CreateSubdirectory(@"MyBar\MyQaaz");

Console.WriteLine("Создан: {0} ", d.FullName);

**Работа с типом Directory**

Теперь, когда вы увидели тип DirectoryInfo в действии, рассмотрим тип Directory. По большей части члены Directory "дублируют" функциональные возможности, обеспечиваемые членами уровня экземпляра DirectoryInfo. Напомним, однако, что члены Directory возвращают строковые типы, а не строго типизированные объекты FileInfo/DirectoryInfo.

Чтобы проиллюстрировать некоторые функциональные возможности типа Directory, заключительная модификация этого примера отображает имена всех дисков, отображаемых на данном компьютере (для этого применяется метод Directorу.GetLogicalDrives()) и используется статический метод Directory. Delete() для удаления ранее созданных подкаталогов \MyFoo и \MyBar\MyQaaz.

class Program {

 static void Main(string[] args) {

  …

**// Список дисков данного компьютера.**

  string[] drives = Directory.GetLogicalDrives();

  Console.WriteLine("Вот ваши диски:");

  foreach (string s in drives) Console.WriteLine(" -› {0}", s);

**// Удаление созданного.**

  Console.WriteLine("Нажмите ‹Enter› для удаления каталогов");

  try {

**// Второй параметр сообщает, хотите ли вы**

**// уничтожить подкаталоги**

   Directory.Delete(@"C:\Windows\MyBar", **true**);

  } catch (IOException e) {

   Console.WriteLine(e.Message);

  }

 }

}

**Исходный код**. Проект MyDirectoryApp размещен в подкаталоге, соответствующем главе 16.

**Работа с типом класса DriveInfo**

В .NET 2.0 пространство имен System.IO предлагает класс с именем DriveInfo. Подобно Directory.GetLogicalDrives(), статический метод DriveInfo.GetDrives() позволяет выяснить имена дисков машины. Однако, в отличие от Directory.GetLogicalDrives(), класс DriveInfo обеспечивает множество дополнительной информации (например, информацию о типе диска, свободном пространстве, метке тома и т.д.). Рассмотрите следующий пример программного кода.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Забавы с DriveInfo \*\*\*\*\*\n'');

**// Получение информации о дисках.**

**// Вывод информации о состоянии.**

  foreach(DriveInfo d in myDrives) {

   Console.WriteLine("Имя: {0}", d.**Name**);

   Console.WriteLine("Тип: {0}", d.**DriveType**);

**// Проверка диска.**

   if (d.IsReady) {

    Console.WriteLine("Свободно: {0}", d.**TotalFreeSpace**);

    Console.WriteLine("Формат: {0}", d.**DriveFormat**);

    Console.WriteLine("Метка тома: {0}\n", d.**VolumeLabel**);

   }

  }

  Console.ReadLine();

 }

}

На рис. 16.5 показан вывод, соответствующий состоянию моей машины.

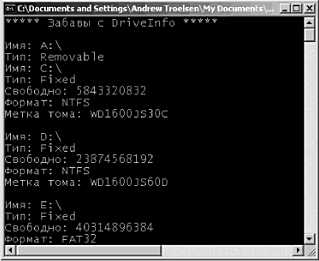


Рис. 16.5. Сбор информации о дисках с помощью DriveInfo

Итак, мы рассмотрели некоторые возможности классов Directory.DirectoryInfo и DriveInfo. Далее вы узнаете, как создавать, открывать, закрывать и уничтожать файлы, присутствующие в каталоге.

**Исходный код.**Проект DriveTypeApp размещен в подкаталоге, соответствующем главе 16.

**Работа с классом FileInfo**

Как показывает пример MyDirectoryApp, класс FileInfo позволяет получить подробные сведения о файлах, имеющихся на вашем жестком диске (время создания, размер, атрибуты и т.д.), а также помогает создавать, копировать, перемещать и уничтожать файлы. Вдобавок к набору функциональных возможностей, унаследованных от FileSystemInfо, класс FileInfo имеет свои уникальные члены, и некоторые из них описаны в табл. 16.4.

**Таблица 16.4.**Наиболее важные элементы FileInfo

|  |  |
| --- | --- |
| **Член** | **Описание** |
| AppendText() | Создает тип StreamWriter (будет описан позже) для добавления текста в файл |
| CopyTo() | Копирует существующий файл в новый файл |
| Create() | Создает новый файл и возвращает тип FileStream (будет описан позже) для взаимодействия с созданным файлом |
| CreateText() | Создает тип StreamWriter, который записывает новый текстовый файл |
| Delete() | Удаляет файл, к которому привязан экземпляр FileInfo |
| Directory | Получает экземпляр каталога родителя |
| DirectoryName | Получает полный путь к каталогу родителя |
| Length | Получает размер текущего файла или каталога |
| MoveTo() | Перемещает указанный файл в новое место, имеет опцию для указания нового имени файла |
| Name | Получает имя файла |
| Open() | Открывает файл с заданными возможностями чтения/записи и совместного доступа |
| OpenRead() | Создает FileStream с доступом только для чтения |
| OpenText() | Создает тип StreamReader (будет описан позже) для чтения из существующего текстового файла |
| OpenWrite() | Создает FileStream с доступом только для записи |

Важно понимать, что большинство членов класса FileInfo возвращает специальный объект ввода-вывода (FileStream, StreamWriter и т.д.), который позволит начать чтение или запись данных в соответствующем файле в самых разных форматах. Мы исследуем указанные типы чуть позже, а пока что давайте рассмотрим различные способы получения дескриптора файла с помощью типа класса FileInfo.

**Метод FileInfо.Create()**

Первая возможность создания дескриптора файла обеспечивается методом FileInfo.Create().

public class Program {

 static void Main(string[] args) {

**// Создание нового файла на диске C.**

  FileInfo f = new FileInfо(@"C:\Test.dat");

  FileStream fs = f.Create();

**// Использование объекта FileStream.…**

**// Закрытие файлового потока.**

  fs.Close();

 }

}

Обратите внимание на то, что метод FileInfo.Create() возвращает тип FileStream, который, в свою очередь, предлагает набор синхронных и асинхронных операций записи/чтения для соответствующего файла. Объект FileStream, возвращенный методом FileInfo.Create(), обеспечивает полный доступ чтения/записи всем пользователям.

**Метод FileInfo.Open()**

Метод FileInfо.Open() можно использовать для того, чтобы открывать существующие файлы и создавать новые с более точными характеристиками, чем при использовании FileInfo.Create(). В результате вызова Open() возвращается объект FileStream. Рассмотрите следующий пример.

static void Main(string[] args) {

 …

**// Создание нового файла с помощью FileInfo.Open().**

 FileInfo f2 = new FileInfo(@"C:\Test2.dat");

 FileStream fs2 = f2.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None);

**// Использование объекта FileStream.…**

**// Закрытие файлового потока.**

 fs2.Close();

}

Эта версия перегруженного метода Open() требует указания трех параметров. Первый параметр задает общий вид запроса ввода-вывода (создание нового файла, открытие существующего файла, добавление данных в файл и т.п.) с помощью перечня FileMode.

public enum FileMode {

**// Дает операционной системе указание создать новый файл.**

**// Если файл уже существует, генерируется System.IO.IOException.**

 CreateNew,

**// Дает операционной системе указание создать новый файл,**

**// Если файл уже существует, он будет переписан.**

 Create,

 Open,

**// Дает операционной системе указание открыть файл,**

**// если он существует, иначе следует создать новый файл.**

 OpenOrCreate,

 Truncate,

 Append

}

Второй параметр, значение из перечня FileAccess, используется для определения характеристик чтения/записи в соответствующем потоке.

public enum FileAccess {

 Read,

 Write,

 ReadWrite

}

Наконец, третий параметр, FileShare, указывает возможности совместного использования файла другими дескрипторами файла. Вот как выглядит соответствующий перечень.

public enum FileShare {

 None,

 Read,

 Write,

 ReadWrite

}

**Методы FileInfo.OpenRead() и FileInfo.OpenWrite()**

Хотя метод FileInfo.Open() и обладает очень гибкими возможностями получения дескриптора файла, класс FileInfo также предлагает члены с именами OpenRead() и OpenWrite(). Как вы можете догадаться, эти методы возвращают должным образом сконфигурированный только для чтения или только для записи тип FileStream, без необходимости указания соответствующих значений перечней.

Подобно FileInfo.Create() и FileInfo.Open(), методы OpenRead() и OpenWrite() возвращают объект FileStream.

static void Main(string[] args) {

 …

**// Получение объекта FileStream с доступом только для чтения.**

 FileInfo f3 = new FileInfo(@"C:\Test3.dat");

 FileStream readOnlyStream = f3.OpenRead();

**// Использование объекта FileStream…**

 readOnlyStream.Close();

**// Получение объекта FileStream с доступом только для записи.**

 FileInfо f4 = new FileInfo(@"C:\Test4.dat");

 FileStream writeOnlyStream = f4.OpenWrite();

**// Использование объекта FileStream…**

 writeOnlyStream.Close();

}

**Метод FileInfo.OpenText()**

Другим членом типа FileInfo, связанным с открытием файлов, является OpenText(). В отличие от Create(), Open(), OpenRead() и OpenWrite(), метод OpenText() возвращает экземпляр типа StreamReader, а не типа FileStream.

static void Main(string[] args) {

 …

**// Получение объекта StreamReader.**

 FileInfo f5 = new FileInfо(@"C:\boot.ini");

 StreamReader sreader = f5.OpenText();

**// Использование объекта StreamReader.…**

 sreader.Close();

}

Чуть позже вы увидите, что тип StreamReader обеспечивает возможность чтения символьных данных из соответствующего файла.

**Методы FileInfo.CreateText() и FileInfo.AppendText()**

И последними интересующими нас на этот момент методами будут CreateText() и AppendText(), которые возвращают ссылку на StreamWriter, как показано ниже.

static void Main(string[] args) {

 …

 FileInfo f6 = new FileInfo(@"C:\Test5.txt");

 StreamWriter swriter = f6.CreateText();

**// Использование объекта StreamWriter….**

 swriter.Close();

 FileInfo f7 = new FileInfo(@"C:\FinalTest.txt");

 StreamWriter swriterAppend = f7.AppendText();

**// Использование объекта StreamWriter…**

 swriterAppend.Close();

}

Вы должны догадаться сами, что тип StreamWriter предлагает способ записи символьных данных в соответствующий файл.

**Работа с типом File**

Тип File предлагает функциональные возможности, почти идентичные возможностям типа FileInfo, но с помощью ряда статических членов. Подобно FileInfo, тип File предлагает методы AppendText(), Create(), CreateText(), Open(), OpenRead(), OpenWrite() и OpenText(). Во многих случаях типы File и

FileStream оказываются взаимозаменяемыми. Так, в каждом из предыдущих примеров вместо FileStream можно использовать тип File.

static void Main(string[] args) {

**// Получение объекта FileStream с помощью File.Create() .**

 FileStream fs = File.Create(@"C:\Test.dat");

 fs.Close();

**// Получение объекта FileStream с помощью File.Open().**

 FileStream fs2 = File.Open(@"C:\Test2.dat", FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None);

 fs2.Close();

**// Получение объекта FileStream с доступом только для чтения.**

 FileStream readOnlyStream = File.OpenRead(@"Test3.dat");

 readOnlyStream.Close();

**// Получение объекта FileStream с доступом только для записи.**

FileStream writeOnlyStream = File.OpenWrite(@"Test4.dat");

 writeOnlyStream.Close();

**// Получение объекта StreamReader.**

 StreamReader sreader = Filе.OpenText(@"C:\boot.ini");

 sreader.Close();

**// Получение нескольких объектов StreamWriter.**

 StreamWriter swriter = File.CreateText(@"C:\Test3.txt");

 swriter.Close();

 StreamWriter swriterAppend = File.AppendText(@"C:\FinalTest.txt");

 swriterAppend.Close();

}

**Новые члены File в .NET 2.0**

В отличие от FileInfo, тип File поддерживает (в .NET 2.0) несколько своих собственных уникальных членов, описания которых приводятся в табл. 16.5. С помощью этих членов можно существенно упростить процессы чтения и записи текстовых данных.

**Таблица 16.5.**Методы типа File

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| ReadAllBytes() | Открывает указанный файл, возращает двоичные данные в виде массива байтов, а затем закрывает файл |
| ReadAllLines() | Открывает указанный файл, возращает символьные данные в виде массива строк, а затем закрывает файл |
| ReadAllText() | Открывает указанный файл, возращает символьные данные в виде System.String, а затем закрывает файл |
| WriteAllBytes() | Открывает указанный файл, записывает массив байтов, а затем закрывает файл |
| WriteAllLines() | Открывает указанный файл, записывает массив строк, а затем закрывает файл |
| WriteAllText() | Открывает указанный файл, записывает символьные данные, а затем закрывает файл |

При использовании этих новых методов типа File для чтения и записи пакетов данных потребуется всего несколько строк программного кода. Более того, каждый из указанных новых членов автоматически закрывает соответствующий дескриптор файла, например:

class Program {

 static void Main(string[] args) {

  string[] myTasks = { "Прочистить сток в ванной", "Позвонить Саше и Сереже", "Позвонить родителям", "Поиграть с ХВох" };

**// Записать все данные в файл на диске C.**

  File.WriteAllLines(@"C:\tasks.txt", myTasks);

**// Прочитать все снова и напечатать.**

  foreach (string task in File.ReadAllLines(@"C:\tasks.txt")) {

   Console.WriteLine("Нужно сделать: {0}", task);

  }

 }

}

Очевидно, когда вы хотите быстро получить дескриптор файла, тип File избавит вас от необходимости ввода нескольких лишних строк. Однако преимущество предварительного создания объекта FileInfo заключается в том, что тогда вы получаете возможность исследовать соответствующий файл с помощью членов абстрактного базового класса FileSystemInfо.

static void Main(string[] args) {

**// Вывод информации о файле boot.ini**

**// с последующим открытием доступа только для чтения.**

 FileInfo bootFile = new FileInfо(@"C:\boot.ini");

 Console.WriteLine(bootFile.CreationTime);

 Console.WriteLine(bootFile.LastAccessTime);

 FileStream readOnlyStream = bootFile.OpenRead();

 readOnlyStream.Close();

}

**Абстрактный класс Stream**

К этому моменту вы уже видели множество способов получения объектов FileStream, StreamReader и StreamWriter, но вам придется еще читать и записывать данные файлов, связанных с этими типами. Чтобы понять, как это делается, нужно ознакомиться с понятием потока. В "мире" ввода-вывода *поток*представляет порцию данных. Потоки обеспечивают общую возможность взаимодействия с*последовательностями байтов,*независимо от того, на устройстве какого вида (в файле, сетевом соединении, принтере и т.п.) они хранятся или отображаются.

Абстрактный класс System.IO.Stream определяет ряд членов, обеспечивающих поддержку синхронного и асинхронного взаимодействия с носителем данных (скажем, с файлом или областью памяти). На рис. 16.6 показано несколько потомков типа Stream.

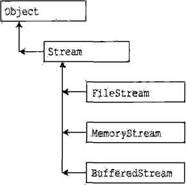


Рис. 16.6. Типы, производные от Stream

**Замечание.**Следует знать, что понятие потока применимо не только к файлам или области памяти. Без сомнения, библиотеки .NET обеспечивают потоковый доступ к сетям и другим связанным с потоками абстракциям.

Напомним, что потомки stream представляют данные в виде "сырого" потока байтов, поэтому работа с потоками может быть весьма непонятной. Некоторые относящиеся к Stream типы поддерживают поиск – этот термин, по сути, означает процесс получения он изменения текущей позиции в потоке. Чтобы понять функциональные возможности, предлагаемые классом Stream, рассмотрите его базовые члены, описанные в табл. 16.6.

**Таблица 16.6.**Абстрактные члены Stream

|  |  |
| --- | --- |
| **Члены** | **Описание** |
| CanRead CanSeek CanWrite | Определяет, поддерживает ли текущий поток чтение, поиск и/или запись |
| Close() | Завершает текущий поток и освобождает все связанные с текущим потоком ресурсы (например, сокеты и дескрипторы файлов) |
| Flush() | Обновляет связанный источник данных или хранилище в соответствии с текущим состоянием буфера, а затем очищает буфер. Если поток не реализует буфер, этот метод не делает ничего |
| Length | Возвращает длину потока в байтах |
| Position | Определяет позицию в текущем потоке |
| Read() ReadByte() | Читает последовательность байтов (или один байт) из текущего потока и сдвигает указатель позиции в соответствии со считанным числом байтов |
| Seek() | Устанавливает указатель в заданную; позицию в текущем потоке |
| SetLength() | Устанавливает длину текущего потока |
| Write() WriteByte() | Записывает последовательность байтов (или один байт) в текущий поток и сдвигает указатель позиции в соответствии со считанным числом байтов |

**Работа с FileStream**

Класс FileStream обеспечивает реализацию абстрактных членов Stream в виде, подходящем для файловых потоков. Это довольно примитивный поток – он может читать или записывать только один байт или массив байтов. На самом деле необходимость непосредственного взаимодействия с членами типа FileStream возникает очень редко. Вы чаще будете использовать различные *упаковщики потоков,*которые упрощают работу с текстовыми данными или типами .NET. Однако для примера давайте поэкспериментируем со средствами синхронного чтения/записи типа FileStream.

Предположим, что мы создали новое консольное приложение FileStreamApp. Нашей целью является запись простого текстового сообщения в новый файл с именем myMessage.dat. Но поскольку FileStream может воздействовать только на отдельные байты, требуется перевести тип System.String в соответствующий массив байтов. К счастью, в пространстве имен System.Text определяется тип Encoding, предлагающий члены, которые выполняют кодирование и декодирование строк и массивов байтов (для подробного описания типа Encoding обратитесь к документации .NET Framework 2.0 SDK).

После выполнения кодирования массив байтов переводится в файл с помощью метода FileStream.Write(). Чтобы прочитать байты обратно в память, необходимо переустановить внутренний указатель позиции потока (с помощью свойства Position) и вызвать метод ReadByte(). Наконец, массив байтов и декодированная строка выводятся на консоль. Вот полный текст соответствующего метода Main().

**// Не забудьте 'использовать' System.Text.**

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с FileStreams \*\*\*\*\*\n");

**// Получение объекта FileStream.**

 FileStream fStream = File.Open(@"C:\myMessage.dat", FileMode.Create);

**// Кодирование строки в виде массива байтов.**

 string msg = "Привет!";

 byte[] msgAsByteArray = Encoding.Default.GetBytes(msg);

**// Запись byte[] в файл.**

 fStream.Write(msgAsByteArray, 0, msgAsByteArray.Length);

**// Переустановка внутреннего указателя позиции потока.**

 fStream.Position = 0;

**// Чтение типов из файла и вывод на консоль….**

 Console.Write("Ваше сообщение в виде массива байтов: ");

 byte[] bytesFromFile = new byte[msgAsByteArray.Length];

 for (int i = 0; i ‹ msgAsByteArray.Length; i++) {

  bytesFromFile[i] = (byte)fStream.ReadByte();

  Console.Write(bytesFromFile[i]);

 }

**// Вывод декодированного сообщения.**

 Console.Write("\nДекодированное сообщение: ");

 Console.WriteLine(Encoding.Default.GetString(bytesFromFile));

**// Завершение потока.**

 fStream.Close();

}

Хотя в этом примере файл данными не заполняется, уже здесь становится очевидным главный недостаток работы с типом FileStream: приходится воздействовать непосредственно на отдельные байты. Другие типы, являющиеся производными от Strеаm, работают аналогично. Например, чтобы записать последовательность байтов в заданную область памяти, можно использовать MemoryStream. Точно так же, чтобы передать массив байтов по сети, вы можете использовать тип NetworkStream.

К счастью, пространство имен System.IO предлагает целый ряд типов "чтения" и "записи", инкапсулирующих особенности работы с типами, производными от Stream.

**Исходный код**. Проект FileStreamApp размещен в подкаталоге, соответствующем главе 16.

**Работа с StreamWriter и StreamReader**

Классы StreamWriter и StreamReader оказываются полезны тогда, когда приходится читать или записывать символьные данные (например, строки). Оба эти типа по умолчанию работают с символами Unicode, однако вы можете изменить эти установки, предоставив ссылку на правильно сконфигурированный объект System.Text.Encoding. Чтобы упростить рассмотрение, предположим, что предлагаемое по умолчавию кодирование в символы Unicode как раз и является подходящим.

Тип StreamReader получается из абстрактного типа TextReader. To же можно сказать и о родственном типе StringReader (он будет обсуждаться в этой главе позже). Базовый класс TextReader обеспечивает каждому из этих "последователей" очень небольшой набор функциональных возможностей, среди которых, в частности, возможность чтения символов из потока и их добавление в поток.

Тип StreamWriter (как и StringWriter, который также будет рассматриваться позже) получается из абстрактного базового класса TextWriter. Этот класс определяет члены, позволяющие производным типам записывать текстовые данные в имеющийся символьный поток. Взаимосвязь между этими новыми типами ввода-вывода показана на рис. 16.7.

Чтобы помочь вам понять возможности записи классов StreamWriter и StringWriter, в табл. 16.7 предлагаются описания основных членов абстрактного базового класса TextWriter.

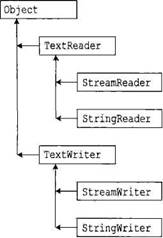


Рис. 16.7. Читатели и писатели

**Таблица 16.7.**Основные члены TextWriter

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Close() | Закрывает записывающий объект и освобождает связанные с ним ресурсы. При этом автоматически очищается буфер |
| Flush() | Очищает все буферы текущего записывающего объекта с тем, чтобы все данные буфера были записаны на соответствующее устройство, но не закрывает сам записывающий объект |
| NewLine | Указывает константу обрыва строки для производного класса записывающего объекта. По умолчанию признаком обрыва строки является возврат каретки с переходом на новую строку (\r\n) |
| Write() | Записывает строку в текстовый поток без добавления константы обрыва строки |
| WriteLine() | Записывает строку в текстовый поток с добавлением константы обрыва строки |

**Замечание.**Последние два из указанных в таблице членов класса TextWriter, вероятно, покажутся вам знакомыми. Если вы помните, у типа System.Console есть члены Write() и WriteLine(), записывающие текстовые данные в устройство стандартного вывода. На самом деле свойство Console.In является упаковкой для TextWriter, а свойство Console.Out – для TextReader.

Производный класс StreamWriter обеспечивает подходящую реализацию методов Write(), Close() и Flush() и определяет дополнительное свойство AutoFlush. Это свойство, когда его значение равно true (истина), заставляет StreamWriter при выполнении операции записи записывать все данные. Можно добиться лучшей производительности, если установить для AutoFlush значение false (ложь), поскольку иначе при каждой записи StreamWriter будет вызываться Close().

**Запись в текстовый файл**

Рассмотрим пример работал с типом StreamWriter. Следующий класс создает новый файл reminders.txt с помощью метода File.CreateText(). С помощью полученного объекта StreamWriter в новый файл добавляются определенные текстовые данные, как показано ниже.

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Забавы с StreamWriter/StreamReader \*\*\*\n");

**// Получение StreamWriter и запись строковых данных.**

 StreamWriter writer = File.CreateText("reminders.txt");

 writer.WriteLine("Нe забыть о дне рождения мамы…");

 writer.WriteLine("Не забыть о дне рождения папы…");

 writer.WriteLine("Не забыть о следующих числах:");

 for(int i = 0; i ‹ 10; i++) writer.Write(i + " ");

**// вставка новой строки.**

 writer.Write(writer.NewLine);

**// Закрытие автоматически влечет запись всех оставшихся данных!**

 writer.Close();

 Console.WriteLine("Создан файл и записаны некоторые идеи…");

}

Выполнив эту программу, вы можете проверить содержимое нового файла (рис. 16.8).

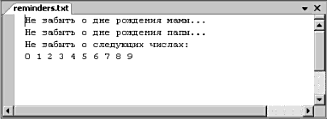


Рис. 16.8. Содержимое вашего файла *\**.txt

**Чтение из текстового файла**

Теперь выясним, как программными средствами читать данные из файла, используя соответствующий тип StreamReader. Вы должны помнить, что этот класс получается из TextReader, функциональные возможности которого описаны в табл. 16.8.

Таблица 16.8. Основные члены TextReader

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Peek() | Возвращает следующий доступный символ без фактического изменения позиции указателя считывающего объекта. Значение -1 указывает позицию, соответствующую концу потока |
| Read() | Читает данные входного потока |
| ReadBlock() | Читает максимальное заданное число символов текущего потока и записывает данные в буфер, начиная с указанного индекса |
| ReadLine() | Читает строку символов из текущего потока и возвращает данные в виде строки (пустая строка указывает EOF – конец файла) |
| ReadToEnd() | Читает все символы, начиная с текущей позиции и до конца потока, и возвращает их в виде одной строки |

Если теперь расширить имеющийся класс MyStreamWriter.Reader, чтобы использовать в нем StreamReader, вы сможете прочитать текстовые данные из файла reminders.txt, как показано ниже.

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Забавы с StreamWriter/StreamReader \*\*\*\n");

 …

**// Теперь чтение данных из файла.**

 Console.WriteLine("Вот ваши идеи:\n");

 StreamReader sr = File.OpenText("reminders.txt");

 string input = null;

 while ((input = sr.ReadLine()) != null) {

  Console.WriteLine(input);

 }

}

Выполнив программу, вы увидите символьные данные из reminders.txt, выведенные на консоль.

**Непосредственное создание типов StreamWriter/StreamReader**

Одной из смущающих особенностей работы с типами из System.IO является то, что часто одних и тех же результатов можно достичь в рамках множества подходов. Например, вы видели, что можно получить StreamWriter из File или из FileInfo, используя метод CreateText(). На самом деле есть еще одна возможность получения StreamWriters и StreamReaders – это непосредственное их создание. Например, наше приложение можно было бы переписать в следующем виде.

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Забавы с StreamWriter/StreamReader \*\*\*\n");

**// Get a StreamWriter and write string data.**

 StreamWriter writer = new StreamWriter("reminders.txt");

 …

**// Now read data from file.**

 StreamReader sr = new StreamReader("reminders.txt");

 …

}

Видеть так много идентичных, на первый взгляд, подходов к реализации ввода-вывода, может быть, немного странно, но имейте в виду, что конечным результатом здесь оказывается гибкость. Так или иначе, вы смогли увидеть, как можно извлекать символьные данные из файлов и помещать их в файлы, используя типы StreamWriter и StreamReader, и теперь мы с вами можем рассмотреть роль классов StringWriter и StringReader.

**Исходный код**. Проект StreamWriterReaderApp размещен в подкаталоге, соответствующем главе 16.

**Работа с типами StringWriter и StringReader**

Используя типы StringWriter и StringReader, вы можете обращаться с текстовой информацией, как с потоком символов в памяти. Это может оказаться полезным тогда, когда необходимо добавить символьную информацию в соответствующий буфер. В следующем примере блок строковых данных записывается в объект StringWriter, а не в файл на локальном жестком диске.

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Забавы с StringWriter/StringReader \*\*\*\n");

**// Создание StringWriter и вывод символьных данных в память.**

 StringWriter strWriter = new StringWriter();

 strWriter.WriteLine("He забыть о дне рождения мамы…");

 strWriter.Close();

**// Получение копии содержимого (сохраненного в строке) и**

**// вывод на консоль.**

 Console.WriteLine("Содержимое StringWriter:\n{0}", strWriter);

}

Ввиду того, что и StringWriter, и StreamWriter получаются из одного и того же базового класса (TextWriter), для них используется приблизительно одинаковая программная логика записи. Однако ввиду самой своей природы, класс StringWriter позволяет извлечь объект System.Text.StringBuilder с помощью метода GetStringBuilder().

static void Main(string[] args) {

 Соnsоlе.WriteLine("\*\*\* Забавы с StringWriter/StringReader \*\*\*\n'');

 …

**// Создание StringWriter и вывод символьных данных в память.**

 StringWriter strWriter = new StringWriter();

 …

**// Получение внутреннего StringBuilder.**

 StringBuilder sb = strWriter.GetStringBuilder();

 sb.Insert(0, "Эй!! ");

 Console.WriteLine("-› {0}", sb.ToString());

 sb.Remove(0, "Эй!! ".Length);

 Console.WriteLine("-› {0}", sb.ToString());

}

Чтобы прочитать данные из символьного потока, используйте соответствующий тип StringReader, который (в соответствии с ожиданиями) функционирует так же, как и родственный ему класс StreamReader. Фактически класс StringReader просто переопределяет наследуемые члены, чтобы обеспечить чтение из блока символьных данных, а не из файла.

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Забавы с StringWriter/StringReader \*\*\*\n");

**// Создание StringWriter и вывод символьных данных в память.**

 StringWriter strWriter = new StringWriter();

 …

**// Чтение данных из StringWriter.**

 StringReader strReader = new StringReader(writer.ToString());

 string input = null;

 while ((input = strReader.ReadLine()) != null) {

  Console.WriteLine(input);

 }

 strReader.Close();

}

**Исходный код.**Проект StringWriterReaderApp размещен в подкаталоге, соответствующем главе 16.

**Работа с BinaryWriter и BinaryReader**

И последним из рассмотренных здесь средств чтения/записи будут BinaryReader и BinaryWriter, которые получаются непосредственно из System.Object. Эти типы позволяют читать и записывать дискретные *типы данных*в соответствующий поток в компактном двоичном формате. Класс BinaryWriter определяет чрезвычайно перегруженный метод Write(), позволяющий поместить тип данных в соответствующий поток. Вдобавок к Write(), класс BinaryWriter предлагает дополнительные члены, позволяющие получить или установить тип, производный от Stream, и обеспечить поддержку прямого доступа к данным (табл. 16.9).

Таблица 16.9. Основные члены BinaryWriter

|  |  |
| --- | --- |
| **Член** | **Описание** |
| BaseStream | Свойство, доступное только для чтения. Обеспечивает доступ к потоку, используемому с объемом BinaryWriter |
| Close() | Метод, завершающий двоичный поток |
| Flush() | Метод, выполняющий очистку двоичного потока |
| Seek() | Метод, устанавливающий указатель позиции в текущем потоке |
| Write() | Метод, записывающий значение в текущий поток |

Класс BinaryReader дополняет функциональные возможности, предлагаемые членами BinaryWriter (табл. 16.10).

**Таблица 16.10.**Основные Члены BinaryReader

|  |  |
| --- | --- |
| **Член** | **Описание** |
| BaseStream | Свойство, доступное только для чтения. Обеспечивает доступ к потоку, используемому с объектом BinaryReader |
| Close() | Метод, завершающий двоичный поток чтения |
| PeekChar() | Метод, возвращающий следующий доступный символ без фактического смещения указателя позиции в потоке |
| Read() | Метод, считывающий заданное множество байтов или символов и запоминающий их во входном массиве |
| ReadXXX() | Класс BinaryReader определяет множество методов ReadXXX(), "захватывающих" следующий тип из потока (ReadBoolean(), ReadByte(), ReadInt32() и т.д.) |

В следующем примере в новый файл \*.dat записывается целый ряд типов данных,

static void Main(string[] args) {

**// Открытие сеанса двоичной записи в файл.**

 FileInfo f = new FileInfo("BinFile.dat");

 BinaryWriter bw = new BinaryWriter(f.OpenWrite());

**// Печать информации о типе BaseStream.**

**// (в данном случае это System.IO.FileStream)**.

 Console.WriteLine("Базовый поток: {0}", bv.BaseStream);

**// Создание порции данных для сохранения в файле.**

 double aDouble = 1234.67;

 int anInt = 34567;

 char[] aCharArray = {*'*A', 'В', 'С'};

**// Запись данных.**

 bw.Write(aDouble);

 bw.Write(anInt);

 bw.Write(aCharArray);

 bw.Close();

}

Обратите внимание на то, что объект FileStream, возвращенный из FileInfo.OpenWrite(), передается конструктору типа BinaryWriter. С помощью такого подхода очень просто выполнить "расслоение" потока перед записью данных. Следует осознавать, что конструктор BinaryWriter способен принять любой тип, производный от Stream (например, FileStream, MemoryStream или BufferedStream). Поэтому, если нужно записать двоичные данные, например, в память, просто укажите подходящий объект MemoryStream.

Для чтения данных из файла BinFile.dat тип BinaryReader предлагает множество опций. Ниже мы используем PeekChar(), чтобы выяснить, имеет ли поток еще данные, и в том случае, когда он их имеет, использовать ReadByte() для получения значения. Обратите внимание на то, что байты форматируются в шестнадцатиричном виде и между ними вставляются семь пробелов.

static void Main(string[] args) {

**// Открытие сеанса двоичной записи в файл.**

 FileInfo f = new FileInfo("BinFile.dat");

 …

**// Чтение данных в виде "сырых" байтов.**

 BinaryReader br = new BinaryReader(f.OpenRead());

 int temp = 0;

 while (br.PeekChar() != -1) {

  Console.Write("{0,7:x}", br.ReadByte());

  if (++temp == 4) {

**// Запись каждых 4 байтов в виде новой строки.**

Console.WriteLine();

   temp = 0;

  }

 Console.WriteLine();

 }

}

**Исходный код**. Проект BinaryWriterReader размещен в подкаталоге, соответствующем главе 16.

Вывод этой программы показан на рис. 16.9.



Рис. 16.9. Чтение байтов из двоичного файла

**Программный мониторинг файлов**

Теперь, когда вы уже знаете возможности различных средств чтения и записи давайте, рассмотрим роль класса FileSystemWatcher. Этот тип может быть исключительно полезен тогда, когда требуется программный мониторинг файлов, имеющихся в данной системе. В частности, с помощью типа FileSystemWatcher можно контролировать любые из действий, указанных в перечне NotifyFilters (значения его членов очевидны, но в случае необходимости более точную информацию можно получить с помощью справочной системы).

publiс enum System.IO.NotifyFilters {

 Attributes, СreationTime,

 DirectoryName, FileName,

 LastAccess, LastWrite,

 Security, Size,

}

Первым делом для работы с типом FileSystemWatcher нужно установить свойство Path, с помощью которого можно указать имя (и место размещения) каталога, содержащего контролируемые файлы, и свойство Filter, с помощью которого определяются расширения контролируемых файлов.

После этого можно указать обработку событий Сhanged, Created и Deleted, которые работают в совокупности с делегатом FileSystemEventHandler. Этот делегат может вызывать любой метод, соответствующий следующему шаблону.

**// Делегат FileSystemEventHandler должен указывать на методы,**

**// имеющие следующую сигнатуру.**

**void**MyNotifacationiHandler(**object** source, **FileSystemEventArgs**e)

Точно так же событие Renamed можно обработать с помощью типа делегата RenamedEventHandler, способного вызывать методы, соответствующие следующему шаблону.

**// Делегат RenamedEventHandler должен указывать на методы,**

**// имеющие следующую сигнатуру*.***

**void**MyNotificationHandler**(object**source**, RenamedEventArgs** e)

Для иллюстрации процесса мониторинга файлов предположим, что мы создали на диске C новый каталог с именем MyFolder, cодержащий различные файлы \*.txt (назовите их так, как пожелаете). Следующее консольное приложение осуществляет мониторинг файлов \*.txt а каталоге MyFоlder и выводит сообщения о событиях, соответствующих созданию, удалению, изменению или переименованию файлов.

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Чудесный монитор файлов \*\*\*\*\*\n");

**// Установка пути для каталога наблюдения.**

 FileSystemWatcher watcher = new FileSystemWatcher();

 try {

  watcher.Path = @"C:\MyFolder";

 } catch(ArgumentException ex) {

  Console.WriteLine(ex.Message);

  return;

 }

**// Установка фильтров наблюдения.**

 watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite | NotifyFilters.FileName | NotifyFilters.DirectoryName;

**// Наблюдение только за текстовыми файлами.**

 watcher.Filter = "\*.txt";

**// Добавление обработчиков событий.**

 watcher.Changed += new FileSystemEventHandler(OnChanged);

 watcher.Created += new FileSystemEventHandler(OnChanged);

 watcher.Deleted += new FileSystemEventHandler(OnChanged);

 watcher.Renamed += new RenamedEventHandler(OnRenamed);

**// Начало наблюдения за каталогом.**

 watcher.EnableRaisingEvents = true;

**// Ожидание сигнала пользователя для выхода из программы.**

 Console.WriteLine(@"Нажмите 'q' для выхода из приложения.");

 while(Console.Read() != 'q');

}

Следующие два обработчика событий просто выводят информацию о модификации текущего файла.

static void OnChanged(object source, FileSystemEventArgs e) {

**// Уведомление об изменении, создании или удалении файла.**

 Console.WriteLine("Файл {0} {1}!", e.FullPath, e.ChangeType);

}

static void OnRenamed(object source, RenamedEventArgs e) {

**// Уведомление о переименовании файла.**

 Console.WriteLine("Файл {0} переименован в\n{1}",

 e.OldFullPath, e.FullPath);

}

Чтобы проверить работу этой программы, запустите приложение и откройте Проводник Windows. Попытайтесь переименовать, создать, удалить файлы \*.txt в MyFolder или выполнить с ними какие-то другие действия, вы увидите, что консольное приложение реагирует на эти действия выводом различной информации о состоянии текстовых файлов (рис. 16.10).

**Исходный код**. Проект MyDirectoryWatcher размещен в подкаталоге, соответствующем главе 16.

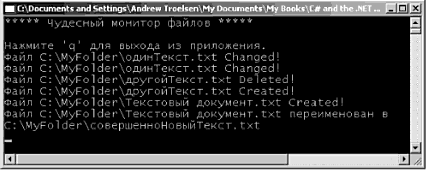


Рис. 16.10. Наблюдение за текстовыми файлами

**Асинхронный файловый ввод-вывод**

В завершение нашего обзора пространства имен System.IO давайте выясним, как осуществляется асинхронное взаимодействие с типами FileStream. Один из вариантов поддержки асинхронного взаимодействия в .NET вы уже видели при рассмотрении многопоточных приложений (см. главу 14). Ввиду того, что ввод-вывод может занимать много времени, все типы, производные от System.IO.Stream, наследуют множество методов, разрешающих асинхронную обработку данных. Как и следует ожидать, эти методы работают в связке с типом IAsyncResult.

public abstract class System.IO.Stream: MarshalByRefObject, IDisposable {

…

 public virtual IAsyncResult BeginRead(byte[] buffer, int offset, int count, AsyncCallback callback, object state);

 public virtual IAsyncResult BeginWrite(byte[] buffer, int offset, int count, AsyncCallback callback, object state);

 public virtual int EndRead(IAsyncResult asyncResult); public virtual void EndWrite(IAsyncResult asyncResult);

}

Работа с асинхронными возможностями типов, производных от System. IO.Stream, аналогична работе с асинхронными делегатами и асинхронными удаленными вызовами методов. Маловероятно, что асинхронный подход может существенно улучшить доступ к файлам, но есть большая вероятность того, что от асинхронной обработки получат выгоду другие потоки (например, использующие сокеты). Так или иначе, следующий пример иллюстрирует подход, в рамках которого вы можете асинхронно взаимодействовать с типом FileStream.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("Старт первичного потока, ThreadID = {0}", Thread.CurrentThread.GetHashCode());

**// Следует использовать этот конструктор, чтобы получить**

**// FileStream с асинхронным доступом для чтения и записи.**

  FileStream fs = new FileStream('logfile.txt", FileMode.Append, FileAccess.Write, FileShare.None, 4096, true);

  string msg = "это проверка";

  byte[] buffer = Encoding.ASCII.GetBytes(msg);

**// Начало асинхронной записи.**

**// По окончании вызывается WriteDone.**

**// Объект FileStream передается методу обратного вызова,**

**// как информация состояния.**

  fs.BeginWrite(buffer, 0, buffer.Length, new AsyncCallback(WriteDone), fs);

 }

 private static void WriteDone(IAsyncResult ar) {

  Console.WriteLine("Метод AsyncCallback для ThreadID = {0}", Thread.CurrentThread.GetHashCode());

  Stream s = (Stream)ar.AsyncState;

  s.EndWrite(ar);

  s.Close();

 }

}

Единственным заслуживающим внимания моментом (при условии, что вы помните основные особенности использования делегатов!) в этом примере является то, что для разрешения асинхронного поведения типа FileStream вы должны использовать специальный конструктор (который здесь и используется). Последний параметр System.Boolean (если он равен true) информирует объект FileStream о том, что соответствующие операции должны выполняться во вторичном потоке.

**Исходный код.**Проект AsynсFileStream размещен в подкаталоге, соответствующем главе 16.

**Резюме**

Эта глава начинается с рассмотрения типов Directory(Info) и File(Info) (а также нескольких новых членов типа File, появившихся в .NET 2.0). Вы узнали о том, что эти классы позволяют работать с физическими файлами или каталогами на жестком диске. Затем был рассмотрен ряд типов (в частности, FileStream), полученных из абстрактного класса Stream. Поскольку типы, производные от Stream, работают с потоком "сырых" байтов, пространство имен System.IO предлагает множество типов ввода-вывода (StreamWriter, StringWriter, BinaryWriter и т.п.), упрощающих процесс.

В процессе обсуждения был также рассмотрен новый тип .NET 2.0 DriveType, вы узнали о том, как контролировать файлы с помощью типа FileSystemWatcher и как взаимодействовать с потоками в асинхронном режиме.

**ГЛАВА 17. Сериализация объектов**

Из главы 16 вы узнали о функциональных возможностях, предоставленных пространством имея System.IO. Было показано, что это пространство имен содержит множество типов ввода-вывода, которые могут использоваться для чтения и сохранения данные в соответствий с заданными параметрами размещения (иди заданным форматом). В этой главе будет рассмотрена родственная тема *сериализации объектов*. С помощью объекта сериализации можно сохранять и восстанавливать состояние объекта в любом производном от System.IO.Stream типе.

Вы сразу согласитесь с тем, что возможность сериализации типов играет ключевую роль при копировании объектов на удаленную машину (этот процесс будет темой обсуждения следующей главы). Однако следует также понимать, что сериализация оказывается полезной и сама по себе, и она, скорее всего, будет играть свою роль во многих ваших .NET-приложениях (как распределенных, так и обычных), В этой главе мы обсудим различные аспекты схемы сериализации .NET, включая множество новых атрибутов, появившихся с выходом .NET 2.0 и позволяющих выполнять пользовательскую настройку соответствующего процесса.

**Основы сериализации объектов**

Термин *сериализация*означает процесс переноса состояния объекта в поток, Соответствующая сохраненная последовательность данных содержит всю информацию, необходимую для реконструкции объекта, если в дальнейшем возникает необходимость в его использовании. С помощью такой технологии очень просто сохранять огромные объемы данных (в самых разных форматах). Во многих случаях сохранение данных приложения с помощью сервиса сериализации оказывается намного менее неуклюжим, чем прямое использование средств чтения/записи, предлагаемых в рамках пространства имен System.IO.

Предположим, например, что вы создали приложение с графическим интерфейсом и хотите обеспечить конечным пользователям возможность сохранить информацию об их предпочтениях. Для этого вы можете определить класс (например, с именем UserPrefs), инкапсулирующий, скажем, 20 полей данных. Если использовать тип System.IO.BinaryWriter, вам придется *вручную*сохранять каждое поле объекта UserPrefs. А когда вы захотите загрузить данные из соответствующего файла обратно в память, вам придется использовать System.IO.BinaryReader и (снова *вручную)*прочитать каждое значение, чтобы сконфигурировать новый объект UserPrefs.

Это, конечно, выполнимо, но вы можете сэкономить себе немало времени, просто указав для класса UserPrefs атрибут [Serializable]. В этом случае для сохранения полного состояния объекта достаточно будет нескольких строк программного кода.

static void Main(string[] args) {

**// Предполагаем, что для UserPrefs**

**// указано [Serializable]**.

 UserPrefs userData = new UserPrefs();

 userData.WindowColor = "Yellow";

 userData.FontSize = "50";

 userData.IsPowerUser = false;

**// Теперь сохраним объект в файле user.dat.**

 BinaryFormatter binFormat = new BinaryFormatter();

 Stream fStream = new FileStream("user.dat", FileMode.Create, FileAccess.Write, FileShare.None);

 binFormat.Serialize(fStream, userData); fStream.Close();

 Console.ReadLine();

}

Сохранять объекты с помощью средств сериализации .NET очень просто, но процессы, происходящие при этом в фоновом режиме, оказываются весьма сложными. Например, когда объект сохраняется в потоке, все соответствующие данные (базовые классы, вложенные объекты и т.п.) автоматически сохраняются тоже. Таким образом, при попытке выполнить сериализацию производного класса в этом процессе будет задействована вся цепочка наследования.

Как вы сможете убедиться позже, множество взаимосвязанных объектов можно представить в виде объектного графа. Сервис сериализации .NET позволяет сохранить и объектный граф, причем в самых разных форматах. В предыдущем примере программного кода использовался тип BinaryFormatter, поэтому состояние объекта UserPrefs сохранялось в компактном двоичном формате. Если использовать другие типы, то объектный граф можно сохранить в формате SOAP (Simple Object Access Protocol – простой протокол доступа к объектам) или в формате XML. Эти форматы могут быть полезны тогда, когда необходимо гарантировать, что ваши сохраненные объекты легко перенесут "путешествие" через операционные системы, языки и архитектуры.

Наконец, следует понимать, что объектный граф можно сохранить в *любом*производном от System.IO.Stream типе. В предыдущем примере объект UserPrefs сохранялся в локальном файле с помощью типа FileStream. Но если бы требовалось сохранить объект в памяти, следовало бы использовать тип MemoryStream. Это необходимо для того, чтобы последовательность данных корректно представляла состояния объектов соответствующего графа.

**Роль объектных графов**

Как уже упоминалось, при сериализации объекта среда CLR учитывает состояния всех связанных объектов. Множество связанных объектов представляется *объектным графом.*Объектные графы обеспечивают простой способ учета взаимных связей в множестве объектов, и не обязательно, чтобы эти связи в точности проецировались в классические связи объектно-ориентированного программирования (такие как отношения старшинства, и подчиненности), хотя они моделируют эту парадигму достаточно хорошо.

Каждому объекту в объектном графе назначается уникальное числовое значение. Следует иметь в виду, что эти числовые значения, приписываемые членам в объектном графе, произвольны и не имеют никакого смысла вне графа. После назначения всем объектам числового значения объектный граф может начать запись множества зависимостей каждого объекта.

Для примера предположим, что вы создали множество классов, моделирующих типы автомобилей (а что же еще?). Вы имеете базовый класс, названный Car (автомобиль), который "имеет" Radio (радио). Другой класс, JamesBondCar (автомобиль Джеймса Бонда), расширяет базовый тип Car. На рис. 17.1 показан возможный объектный граф, моделирующий указанные взаимосвязи.

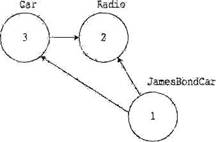


Рис. 17.1 Простой объектный граф

При чтении объектных графов дли соединяющих стрелок вы можете использовать выражения "зависит от" и "ссылается на". Поэтому на рис. 17.1 вы можете видеть, что класс Car ссылается на класс Radio (в силу отношения локализации, "has-a"), а класс JamesBondCar ссылается на Car (в силу отношения подчиненности, "is-а") и на Radio (в силу того, что соответствующий защищенный член-переменная данным классом наследуется).

Конечно, для представления графа связанных объектов среда CLR картины в памяти не рисует. Вместо этого взаимосвязи, указанные в диаграмме, представляются математической формулой, которая выглядит примерно так.

[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]

Проанализировав эту формулу, вы снова увидите, что объект 3 (Car) имеет зависимость в отношения объекта 2 (Radio). Объект 2 (Radio) является "индивидуалистом", которому никто не требуется. Наконец, объект 1 (JamesBondCar) имеет зависимость в отношении как объекта 3, так и объекта 2. В любом случае, когда выполняется сериализация или реконструкция экземпляра JamesBondCar, объектный граф дает гарантию того, что типы Radio и Car тоже будут участвовать в процессе.

Приятной особенностью процесса сериализации является то, что граф, изображающий взаимосвязи ваших объектов, создается в фоновом режиме и автоматически. Позже, в этой же главе, вы убедитесь, что при желании вы все же можете участвовать в построении такого объектного графа.

**Конфигурирование объектов для сериализации**

Чтобы сделать объект доступным сервису сериализации .NET, достаточно пометить каждый связанный класс атрибутом [Serializable]. И это все (правда!). Если вы решите, что некоторые члены данного класса не должны (или, возможно, не могут) участвовать в процессе сериализации, обозначьте соответствующие поля атрибутом [NonSerialized]. Это может быть полезно тогда, когда в классе, предназначенном для сериализации, есть члены-переменные, которые запоминать не нужно (например, фиксированные или случайные значения, динамические данные и т.п.), и вы хотите уменьшить размеры сохраняемого графа.

Для начала вот вам класс Radio, обозначенный атрибутом [Serializable], за исключением одной переменной (radioID), которая помечена атрибутом [NonSerialized], и поэтому не будет сохраняться в указанном потоке данных.

[Serializable]

public class Radio {

 public bool hasTweeters;

 public bool hasSubWoofers;

 public double[] stationPresets;

 [NonSerialized]

 public string radioID = "XF-552RR6";

}

Класс JamesBondCar и базовый класс Car, также обозначенные атрибутом [Serializable], определяют следующие поля данных.

[Serializable]

public class Car {

 public Radio theRadio = new Radio();

 public bool isHatchBack;

}

[Serializable]

public class JamesBondCar: Car {

 public bool canFly;

 public bool canSubmerge;

}

Следует знать о том, что атрибут [Serializable] не наследуется. Таким образом, если вы получаете класс из типа, обозначенного атрибутом [Serializable], дочерний класс тоже следует обозначить атрибутом [Serializable], иначе он при сериализации сохраняться не будет. На самом деле все объекты в объектном графе должны обозначаться атрибутом [Serializable]. При попытке с помощью BinaryFormatter или SoapFormatter выполнить сериализацию объекта. не подлежащего сериализации, в среде выполнения генерируется исключение SerializationException.

**Открытые поля, приватные поля и открытые свойства**

Заметим, что в указанных выше классах поля данных были определены открытыми только для того, чтобы упростить пример. Конечно, с точки зрения объектно-ориентированного подхода предпочтительнее использовать приватные данные, доступные через открытые свойства. Также для простоты не было определено никаких пользовательских конструкторов для этих типов, поэтому все их поля данных, не получившие начальных значений, получат значения, предусмотренные по умолчанию.

"Отодвинув" принципы объектно-ориентированного программирования в сторону, вы можете спросить, какие именно определения полей данных ожидают "видеть" различные средства форматирования, при отправке этих данных в поток. Ответ здесь зависит от многого. Если вы сохраняете объект с помощью BinaryFormatter, то определения не имеют абсолютно никакого значений. Этот тип предназначен дан сохранения *всех* предназначенных для сериализации полей типа, независимо от того, являются ли они общими полями, приватными полями или приватными полями, доступными через свойства типа. Однако ситуация оказывается совершенно иной, если вы используете тип XmlSerializer или тип SoapFormatter. Эти типы выполняют сериализацию только открытых полей данных и приватных данных, доступных через открытые свойства.

Напомним, однако, что если имеются поля данных, которые вы не хотите сохранять в объектном графе, вы можете селективно использовать для них атрибут [NonSerialized], как это сделано со строковым полем типа Radio.

**Выбор формата сериализации**

После конфигурации типов для участия в схеме сериализации .NET следующим шагом является выбор формата, который должен использоваться при сохранении объектного графа. В .NET 2.0 вы имеете на выбор три варианта.

• BinaryFormatter

• SoapFormatter

• XmlSerializer

Тип BinaryFormatter выполняет сериализацию объектного графа в поток, используя компактный двоичный формат. Этот тип определен в рамках пространства имен System.Runtime.Serialization.Formatters.Binary, являющегося частью mscorlib.dll. Таким образом, для сериализации объектов с использованием двоичного формата нужно только указать (в C#) следующую директиву using.

**// Получение доступа к BinaryFormatter из mscorlib.dll**.

using System.Runtime.Serialization.Formatter.Binary;

Тип SoapFormatter представляет граф в виде сообщения SOAP. Этот тип определен в пространстве имен System.Runtime.Serialization.Formatters.Soap, которое содержится *в отдельном компоновочном блоке.*Поэтому, чтобы представить объектный граф в формате сообщения SOAP, вы должны добавить ссылку на System.Runtime.Serialization.Formatters.Soap.dll и указать (в C#) следующую директиву using.

**// Должна быть указана ссылка**

**// на System.Runtime.Serialization.Formatters.Soap.dll!**

using System.Runtime.Serialization.Formatters.Soap;

Наконец, чтобы сохранить объектный граф в формате документа XML, нужно указать ссылку на пространство имен System.Xml.Serialization, которое также определено в отдельном компоновочном блоке – System.Xml.dll. Поскольку все шаблоны проектов в Visual Studio 2005 автоматически ссылаются на System.Xml.dll, вам нужно просто использовать следующее пространство имен.

**// Определено в System.Xml.dll.**

using System.Xml.Serialization;

**Интерфейсы IFormatter и IRemotingFormatter**

Независимо от того, какой формат вы выберете для использования, все они получаются прямо из System.Object и поэтому не могут иметь общего набора членов, наследуемого от какого-либо базового класса сериализации. Однако типы BinaryFormatter и SoapFormatter имеют общее множество членов по причине реализации интерфейсов IFormatter и IRemotingFormatter (тип XmlSerializer не реализует ни одного из них).

Интерфейс System.Runtime.Serialization.IFormatter определяет базовые методы Serialize() и Deserialize(), выполняющие основную работу по перемещению объектных графов в поток и из него. Кроме этих членов, IFormatter определяет несколько свойств, которые используются реализующим типом в фоновом режиме.

public interface **IFormatter**{

 SerializationBinder Binder { get; set; }

 StreamingContext Context { get; set; }

 ISurrogateSelector SurrogateSelector { get; set; }

 object Deserialize(System.IO.Stream serializationStream);

 void Serialize(System.IO.Stream serializationStream, object graph);

}

Интерфейс System.Runtime.Remoting.Messaging.IRemotingFormatter (который используется в .NET на уровне удаленного взаимодействия) предлагает перегруженные члены Serialize() и Deserialize(), более подходящие для использования в распределенных операциях. Заметьте, что IRemotingFormatter получается из более общего интерфейса IFormatter.

public interface **IRemotingFormatter**: IFormatter {

 object Deserialize(Stream serializationStream, HeaderHandler handler);

 void Serialize(Stream serializationStream, object graph, Header[] headers);

}

Хотя в большинстве операций сериализации вам, возможно, и не придется взаимодействовать с этими интерфейсами непосредственно, не забывайте о том, что интерфейсный полиморфизм дозволяет использовать экземпляр BinaryFormatter или SoapFormatter по ссылке на IFormatter. Таким образом, чтобы построить метод, выполняющий сериализацию объектного графа с помощью любого из этих классов, вы можете использовать следующий программный код.

static void SerializeObjectGraph**(IFormatter**itfFormat, Stream destStream, object graph) {

 itfFormat.Serialize(destStream, graph);

}

**Выбор формата и точность типов**

Очевидно, сутью различий указанных трех форматов является то, как именно объектный граф переводится в поток (в двоичном формате, формате SOAP или "чистом" XML). Но следует знать и о нескольких более "утонченных" различиях, особенно в отношении того, насколько различные форматы гарантируют точность *типа*. При использовании типа BinarуFormatter будут сохраняться не только поля данных объектов из объектного графа, но и абсолютное имя каждого типа, а также полное имя определяющего тип компоновочного блока. Эти дополнительные элементы данных делают BinaryFormatter идеальный выбором, когда вы хотите передать объекты по значению (например, как полную копию) за границы машины (см. главу 18). Как уже отмечалось, чтобы достичь такого уровня точности, BinaryFormatter учитывает все поля данных типа (как открытые, так и приватные).

Но SoapFormatter и XmlSerializer, с другой стороны, не пытаются сохранить тип абсолютно точно, поэтому они не записывают абсолютные имена типов и компоновочных блоков, а сохраняют только открытые поля данных и открытые свойства. На первый взгляд, это кажется ограничением, но реальная причина этого скрывается в открытой природе представления данных XML. Если вы хотите сохранить объектные графы так, чтобы они могли использоваться в любой операционной системе (Windows XP, ОС Маc X, различные вариации \*nix), в рамках любого каркаса приложений (.NET. J2EE, COM и т.д.) и любом языке программирования, нет необходимости поддерживать абсолютную точность, поскольку у вас нет гарантии, что все возможные получатели смогут понять типы данных, специфичные для .NET. В этом случае идеальным выбором являются SoapFormatter и XmlSerializer, гарантирующие наиболее широкую доступность сохраненного объектного графа.

**Сериализация объектов с помощью BinaryFormatter**

Чтобы показать, как сохранить экземпляр JamesBondCar в физическом файле, давайте используем тип BinaryFormatter. Подчеркнем снова, что двумя ключевыми методами типа BinaryFormatter являются Serialize() и Deserialize().

• Serialize(). Сохраняет объектный граф в указанном потоке в виде последовательности байтов.

• Deserialize(). Преобразует сохраненную последовательность байтов в объектный граф.

Предположим, что мы создали экземпляр JamesBondCar, изменили в нем некоторые данные и хотим сохранить этот "шпиономобиль" в файле \*.dat. Первой нашей задачей является создание самого файла \*.dat. Это можно сделать с помощью создания экземпляра типа System.IO.FileStream (см. главу 16). Создайте экземпляр BinaryFormatter и передайте ему FileStream и объектный граф для сохранения.

using System.Runtime.Serialization.Formatters.Binary; using System.IO;

 …

 static void Main (string[] args) {

 Console.WriteLine("\*\*\* Забавы с сериализацией объектов \*\*\*\n");

**// Создание JamesBondCar и установка данных состояния.**

 JamesBondCar jbc = new JamesBondCar();

 jbc.canFly = true;

 jbc.canSubmerge = false;

 jbc.theRadio.statio.nPresets = new double[]{89.3, 105.1, 97.1};

 jbc.theRadio.hasTweeters = true;

**// Сохранение объекта в файл CarData.dat в двоичном формате.**

 BinaryFormatter binFormat = new BinaryFormatter();

 Stream fStream = new FileStream("CarData.dat", FileMode.Create, FileAccess.Write, FileShare.None);

 binFormat.Serialize(fStream, jbc);

 fStream.Close();

 Console.ReadLine();

}

Как видите, метод BinaryFormatter.Serialize() отвечает за компоновку объектного графа и передачу соответствующей последовательности байтов некоторому типу, производному от Stream. В данном случае таким потоком является физический файл. Однако можно выполнять сериализацию объектных типов в любой производный от Stream тип, например в память, поскольку MemoryStream тоже является потомком типа Stream.

**Реконструкция объектов с помощью BinaryFormatter**

Теперь предположим, что вы хотите прочитать сохранённые данные JamesBondCar из двоичного файла назад в объектную переменную. Программно открыв CarData.dat (с помощью метода OpenRead()), вызовите метод Deserialize() объекта BinaryFormatter. Метод Deserialize() возвращает общий тип System.Object, поэтому вам придется выполнить явное преобразование, как показано ниже.

static void Main(string[] args) {

 …

**// Чтение JamesBondCar из двоичного файла.**

 fStream = File.OpenRead("CarData.dat");

 JamesBondCar carFromDisk = (JamesBondCar)binFormat.Deserialize(fStream);

 Console.WriteLine("Может ли машина летать?: {0}", carFromDisk.canFly);

 fStream.Close();

 Console.ReadLine();

}

Обратите внимание на то, что при вызове Deserialize() методу передается производный от Stream тип, указывающий место хранения объектного графа (в данном случае это файловый поток). Так что проще уже некуда. По сути, сначала нужно обозначить атрибутом [Serializable] все классы, предназначенные для сохранения в потоке. После этого нужно использовать тип BinaryFormatter, чтобы передать объектный граф в двоичный поток и извлечь его оттуда. Вы можете увидеть двоичный образ, представляющий экземпляр JamesBondCar (рис. 17.2).

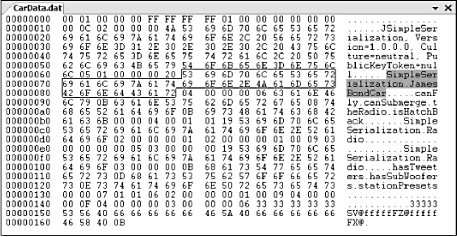


Рис. 17.2. Сериализация JamesBondCar с помощью BinaryFormatter

**Сериализация объектов с помощью SoapFormatter**

Следующим вариантом является тип SoapFormatter. Тип SoapFormatter сохраняет объектный граф в сообщении SOAP (Simple Object Access Protocol – простой протокол доступа к объектам), что делает этот вариант форматирования прекрасным выбором при передаче объектов средствами удаленного взаимодействия по протоколу HTTP. Если вы не знакомы со спецификациями SOAP, не волнуйтесь. В сущности, SOAP определяет стандартный процесс, с помощью которого можно вызывать методы не зависящим от платформы и ОС способом (мы рассмотрим SOAP чуть более подробно в последней главе этой книги при обсуждении Web-сервисов XML).

В предположении о том, что вы установили ссылку на компоновочный блок System.Runtime.Serialization.Formatters.Soap.dll, можно реализовать сохранение и восстановление JamesBondCar в формате сообщения SOAP с помощью замены BinaryFormatter на SoapFormatter. Рассмотрите следующий программный код, который выполняет сериализацию объекта в локальный файл с именем CarData.soap.

using System.Runtime.Serialization.Formatters.Soap;

…

static void Main(string[] args) {

 …

**// Сохранение объекта в файл CarData.soap в формате SOAP.**

 SoapFormatter soapFormat = new SoapFormatter();

 fStream = new FileStream("CarData.soap", FileMode.Create, FileAccess.Write, FileShare.None);

 soapFormat.Serialize(fStream, jbc);

 fStream.Close();

 Console.ReadLine();

}

Как и ранее, здесь просто используются Serialize() и Deserialize() для перемещения объектного графа в поток и восстановления его из потока. Если открыть полученный файл \*.soap, вы увидите в нем элементы XML, представляющие значения JamesBondCar и взаимосвязи между объектами графа (с помощью лексем #ref). Рассмотрите следующий фрагмент XML-кода, соответствующий конечному результату (для краткости здесь опущены указания на пространства имен XML).

‹SOAP-ENV:Envelope xmlns:xsi="…"›

 ‹SOAP-ENV:Body›

  ‹a1:JamesBondCar id="ref-1" xmlns:a1="…"›

   ‹canFly›true‹/canFly›

   ‹canSubmerge›false‹/canSubmerge›

   ‹theRadio href="#ref-3"/›

   ‹isHatchBack›false‹/isHatchBack›

  ‹/a1:JamesBondCar›

  ‹a1:Radio id="ref-3" xmlns:a1="…"›

   ‹hasTweeters›true‹/hasTweeters›

   ‹hasSubWoofers›false‹/hasSubWoofers›

   ‹stationPresets href="ref-4"/›

  ‹/a1:Radio›

  ‹SOAP-ENC:Array id="ref-4" SOAP-ENC:arrayType="xsd:dooble[3]"›

   ‹item›89.3‹/item›

   ‹item›105.1‹/item›

   ‹item›97.1‹/item›

  ‹/SOAP-ENC:Array›

 ‹/SOAP-ENV:Body›

 ‹/SOAP-ENV:Envelope›

**Сериализация объектов с помощью XmlSerializer**

Вдобавок к SOAP и двоичному формату, компоновочный блок System.Xml.dll предлагает третий формат, обеспечиваемый типом System.Xml.Serialization. XmlSerializer который может использоваться для сохранения состояния данного объекта в виде "чистого" XML в противоположность данным XML, упакованным в сообщении SOAP. Работа с этим типом немного отличается от работы с типами SoapFormatter и BinaryFormatter. Рассмотрим следующий программный код.

using Sуstem.Xml.Serialization;

…

static void Main(string[] args) {

 …

**// Сохранение объекта в файл CarData.xml в формате XML.**

 XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar), new Type[] { typeof(Radio), typeof(Car) });

 fStream = new FileStream("CarData.xml", FileMode.Create, FileAccess.Write, FileShare.None);

 xmlFormat.Serialize(fStream, jbc);

 fStream.Close();

 …

}

Здесь главным отличием является то, что тип XmlSerializer требует указания информации о типе соответствующего элемента объектного графа. Обратите внимание на то, что первый аргумент конструктора XmlSerializer определяет корневой элемент XML-файла, а второй аргумент является массивом типов System.Type, содержащих метаданные подчиненных элементов. Если заглянуть в сгенерированный файл CarData.xml, вы увидите следующий XML-код (здесь он приводится в сокращенном виде).

‹?xml version="1.0" encoding="utf-8"?›

 ‹JamesBondCar xmlns:xsi="…"›

  ‹theRadio›

   ‹hasTweeters›true‹/hasTweeters›

   ‹hasSubWoofers›false‹/hasSubwoofers›

   ‹stationPresets›

    ‹double›89.3‹/double›

    ‹double›105.1‹/double›

    ‹double›97.1‹/double›

   ‹/stationPresets›

  ‹/theRadio›

  ‹isHatchBack›false‹/isHatchBack›

  ‹canFly›true‹/canFly›

  ‹canSubmerge›false‹/canSubmerge›

 ‹/JamesBondCar›

**Замечание.**Для XmlSerializer требуется, чтобы все типы в объектном графе, предназначенные для сериализации, поддерживали конструктор, заданный по умолчанию (так что не забудьте добавить его, если вы определили пользовательские конструкторы). Если это условие не будет выполнено, в среде выполнения будет сгенерировано исключение InvalidOperationException.

**Контроль генерируемых XML-данных**

Если у вас есть опыт использования XML-технологий, вы должны хорошо знать о том, что в документе XML очень важно гарантировать соответствие элементов набору правил, обеспечивающих "допустимость" данных. Следует понимать, что "допустимость" XML-документа не связана напрямую с синтаксической правильностью его XML-элементов (например, с требованием о том, что все открываемые элементы должны иметь закрывающие их дескрипторы). Скорее, допустимость документов связана с правилами форматирования (например, поле X должно быть атрибутом и не вложенным элементом), которые обычно задаются XML-схемой или DTD-файлом (файл определения типа документа),

По умолчанию все поля данных типа [Serializable] форматируются, как элементы, а не как XML-атрибуты. Для контроля того, как XmlSerializer компонует генерируемый XML-документ, следует указать для типов [Serializable] дополнительные атрибуты из пространства имен System.Xml.Serialization. В табл. 17.1 представлены некоторые из атрибутов, влияющих на кодирование XML-данных, передаваемых в поток.

**Таблица 17.1.**Атрибуты пространства имен System.Xml.Serialization, связанные с сериализацией объектов

|  |  |
| --- | --- |
| **Атрибут** | **Описание** |
| XmlAttributeAttribute | Член будет сохранен в виде XML-атрибута |
| XmlElementAttribute | Поле или свойство будут сохранены в виде XML-элемента |
| XmlEnumAttribute | Имя элемента перечня |
| XmlRootAttribute | Атрибут, контролирующий формат корневого элемента (пространство имен и имя элемента) |
| XmlTextAttribute | Свойство или поле должно сохраняться в виде XML-текста |
| XmlTypeAtttribute | Имя и пространство имен XML-типа |

Для примера давайте сначала выясним, как поля данных JamesBondCar сохраняются в XML-документе в настоящий момент.

‹?xml version="1.0" encodings="utf-8"?›

‹JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd=http://www.w3.org/2001/XMLSchema›

 …

**‹canFly›true‹/canFly›**

**‹canSubmerge›false*‹/*canSubmerge**›

*‹*/JamesBondCar›

Если вы хотите указать пользовательское пространство имен XML, соответствующее JamesBondCar, и кодировать значения canFly и canSubmerge в виде XML-атрибутов, это можно сделать с помощью изменения определения JamesBondCar в C# следующим образом.

[**Serializablе**, **XmlRoot(Namespace = "http://www.intertechtraining.com")]**

public class JamesBondCar: Car {

 …

**[XmlAttribute]**

 public bool canFly;

**[XmlAttribute]**

 public bool canSubmerge;

}

Это должно дать в результате следующий XML-документ (обратите внимание на открывающий элемент ‹JamesBondCar›).

‹?xml version="1.0" encodin="utf-8"?›

‹JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" **canFly="true" canSubmerge="false"**

**xmlns="http://www.intertechtraining.com"›**

 …

‹/JamesBondCar›"

Конечно, есть множество других атрибутов, которые вы можете использовать для управления процессом генерирования XML-документа с помощью XmlSerializer. Чтобы ознакомиться со всеми опциями, выполните поиск информации о пространстве имен System.Xml.Serialization в документации .NET Framework 2.0 SDK

**Сохранение коллекций объектов**

Теперь вы знаете, как сохранить в потоке отдельный объект, и давайте выясним, как сохранить множество объектов. Заметим, что метод Serialize() интерфейса IFormatter не позволяет указать произвольное число объектов (а только один System.Object). Аналогично, возвращаемым значением Deserialize() тоже является один System.Object.

public interface **IFormatter**{

 …

**object**Deserialize(System.IO.Stream serializationStream);

 void Serialize(System.IO.Stream serializationStream, **object graph**);

}

Напомним, что System.Object фактически представляет весь объектный граф. Поэтому при передаче объекта, обозначенного атрибутом [Serializable] и содержащего другие объекты [Serializable], будет сохранен сразу весь набор объектов. Большинство типов, находящихся в рамках пространства имен System.Collections и System.Collections.Generic, уже обозначены атрибутом [Serializable]. Таким образом, чтобы сохранить набор объектов, просто добавьте этот набор в контейнер (например, в ArrayList или List‹›) и выполните сериализацию полученного объекта в подходящий поток.

Предположим, что в класс JamesBondCar был добавлен конструктор с двумя аргументами, чтобы можно было установить некоторые начальные данные состояния (обратите внимание на то, что конструктор, заданный по умолчанию, был возвращен на место в соответствии с требованиями XmlSerializer),

[Serializable,

XmlRoot(Namespace = "http://www.intartechtraining.com")]

public class JamesBondCar: Car {

 public JamesBondCar(bool skyWorthy, bool seaWorthy) {

  canFly = skyWorthy; canSubmerge = seaWorthy;

 }

**// Для XmlSerializer нужен конструктор, заданный по умолчанию!**

 public JamesBondCar(){}

 …

}

При этом вы сможете сохранить любое число объектов JamesBondCar так.

static void Main(string[] args) {

 …

**// Сохранение объекта List‹› с набором JamesBondCar.**

 List‹JamesBondCar› myCars = new List‹JamesBondCar›();

 myCars.Add(new JamesBondCar(true, true));

 myCars.Add(new JamesBondCar(true, false));

 myCars.Add(new JamesBondCar(false, true));

 myCars.Add(new JamesBondCar(false, false));

 fStream = new FileStream("CarCollection.xml", FileMode.Create, FileAccess.Write, FileShare.None);

 xmlFormat = new XmlSerializer(typeof(List‹JamesBondCar›), new Type[] {typeof(JamesBondCar), typeof(Car), typeof(Radio)});

 xmlFormat.Serialize(fStream, myCars);

 fStream.Close();

 Console.ReadLine();

}

Снова обращаем внимание на то, что по причине использования XmlSerializer требуется указать информацию типа для каждого из объектов, вложенных в корневой объект (которым в данном случае является List‹›). При использовании BinaryFormatter или SoapFormatter программная логика будет еще проще.

statiс void Main (string[] args) {

 …

**// Сохранение объекта List‹›(myCars) в двоичном формате.**

 list‹JamesBondCar› myCars = new List‹JamesBondCar›();

 …

 BinaryFormatter binFormat = new BinaryFormatter();

 Stream fStream = new FileStream("AllMyCars.dat", FileMode.Create, FileAccess.Write, FileShare.None);

 binFormat.Serialize(fStream, myCars);

 fStream.Close();

 Console.ReadLine();

}

Превосходно! К этому моменту вам должно быть понятно, как использовать сервис сериализации объектов для упрощения процесса сохранения и восстановления данных вашего приложения. Теперь давайте выясним, как использовать пользовательские настройки процесса сериализации.

Исходный код. Проект SimpleSerialize размещен в подкаталоге, соответствующем главе 17.

**Настройка процесса сериализации**

В большинстве случаев типовая схема сериализации, предлагаемая платформой .NET, будет именно тем, что требуется. Тогда нужно просто применить атрибут [Serializable] и передать объектный граф выбранному средству форматирования. Но в некоторых случаях может потребоваться корректировка того, как обрабатывается объектный граф в процессе сериализации. Например, в соответствии с внутренними правилами вашей компании все поля данных должны сохраняться в формате верхнего регистра или, возможно, вы хотите добавить в поток дополнительные элементы данных, которые не проецируются непосредственно в поля сохраняемого объекта (это могут быть штампы времени, уникальные имена или что-то иное).

Для непосредственного участия в управлении процессом сериализации объектов пространство имен System.Runtime.Serialization предлагает специальные типы. В табл. 17.2 описаны те из них, о которых вам следует знать.

**Таблица 17.2.**Основные типы пространства имен System.Runtime.Serialization

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| ISerializable | В .NET 1.1 реализация этого интерфейса была наиболее предпочтительным методом пользовательской сериализации объектов. В .NET 2.0 для настройки параметров процесса сериализации предпочтительнее использовать новое множество атрибутов (они будут описаны чуть позже) |
| ObjectIDGenerator | Тип, генерирующий идентификаторы элементов объектного графа |
| OnDeserializedAttribute | Атрибут .NET 2.0, позволяющий указать метод, который вызывается сразу же после выполнения реконструкции объекта |
| OnDeserializingAttribute | Атрибут .NET 2.0, позволяющий указать метод, который вызывается в процессе выполнения реконструкции объекта |
| OnSerializedAttribute | Атрибут .NET 2.0, позволяющий указать метод, который вызывается сразу же после выполнения сериализации объекта |
| OnSerializingAttribute | Атрибут .NET 2.0, позволяющий указать метод, который вызывается в процессе сериализации |
| OptionalFieldAttribute | Атрибут .NET 2.0, позволяющий указать поле типа, которое может отсутствовать в указанном потоке |
| SerializationInfo | По сути, этот класс является "чемоданом свойств", содержащим пары имен и значений, представляющих состояние объекта в процессе сериализации |

**Более глубокий взгляд на сериализацию объектов**

Перед тем как рассмотреть различные способы настройки параметров процесса сериализации, было бы полезно выяснить, что при этом происходит "за кулисами". Когда тип BinaryFormatter выполняет сериализацию объектного графа, этот тип отвечает за передачу в указанный поток следующей информации:

• абсолютных имен объектов графа (например, MyApp.JamesBondCar);

• имени компоновочного блока, определяющего объектный граф (например, MyApp.exe);

• экземпляра класса SerializationInfo, содержащего все данные, поддерживаемые членами объектного графа.

В процессе реконструкции объекта тип BinaryFormatter использует ту же информацию, извлеченную из соответствующего потока, для построения абсолютно точной копии объекта.

**Замечание**. Напомним, что SoapFormatter и XmlSerializer не сохраняют абсолютное имя типа и имя определяющего компоновочного блока. Эти типы заботятся только о сохранении открытых полей данных.

Общую картину можно представлять в виде диаграммы, показанной на рис. 17.3.

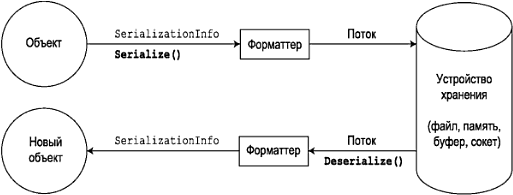


Рис. 17.3 Схема процесса сериализации

Кроме перемещения необходимых данных в поток и извлечения их из потока, средство форматирования (форматтер) анализирует члены объектного графа на наличие следующих элементов инфраструктуры.

• Выясняется, обозначен ли объект атрибутом [Serializable]. Если нет, то генерируется исключение SerializationException.

• Если объект обозначен атрибутом [Serializable], то выясняется, реализует ли объект интерфейс ISerializable. Если да, то для объекта вызывается GetObjectData().

• Если объект не реализует ISerializable, используется типовой процесс сериализации, сохраняются все поля, не обозначенные атрибутом [NonSerialized].

Вдобавок к выявлению поддержки типом интерфейса ISerializable, форматтеры (в .NET 2.0) отвечают также за выявление поддержки соответствующими типами членов, обозначенных атрибутами [OnSerializing], [OnSerialized], [OnDeserializing] или [OnDeserialized]. Роль этих атрибутов будет обсуждаться позже, a пока что мы рассмотрим роль ISerializable.

**Настройка параметров сериализации с помощью ISerializable**

Объекты, обозначаемые атрибутом [Serializable], имеют возможность реализовать интерфейс ISerializable. В этом случае вы можете "участвовать" в процессе сериализации, выполняя любое предварительное или последующее форматирование данных. Указанный интерфейс очень прост, поскольку он определяет единственный метод, GetObjectData().

**// Для настройки процесса сериализации реализуйте ISerializable.**

public interface ISerializable {

 void GetObjectData(SerializationInfo info, StreamingContext context);

}

Метод GetObjectData() вызывается форматтером в процессе сериализации автоматически. Реализация этого метода предоставляет через входной параметр SerializationInfo серию пар имен и значений, которые (обычно) соответствуют полям данных того объекта, который следует сохранить. Тип SerializationInfo определяет перегруженный метод AddValue(), имеющий множество вариаций, а также небольшой набор свойств, которые позволяют читать и устанавливать имя типа, имя определяющего компоновочного блока и значение счетчика членов. Вот фрагмент соответствующего программного кода.

public sealed class SerializationInfo: object {

 public SerializationInfo(Type type, IFormatterConverter converter);

 public string AssemblyName { get; set; }

 public string FullTypeName { get; set; }

 public int MemberCount { get; }

 public void AddValue(string name, short value);

 public void AddValue(string name, UInt16 value);

 public void AddValue(string name, int value);

 …

}

Типы, реализующие интерфейс ISerializable, должны также определять специальный конструктор в соответствии со следующим шаблоном.

**// Следует предложить пользовательский конструктор следующего вида,**

**// чтобы среда выполнения могла установить состояние вашего объекта.**

[Serializable]

class SomeClass: ISerializable {

 private SomeClass(SerializationInfo si, StreamingContext ctx) {…}

 …

}

Обратите внимание на то, что для области видимости этого конструктора указано private. Это вполне допустимо, поскольку форматтер получает доступ к этому члену независимо от его видимости. Эти специальные конструкторы чаще всего обозначаются как приватные, чтобы обеспечить невозможность случайного создания объекта пользователем объекта с помощью такого конструктора. Заметьте, что первый параметр этого конструктора является (как и ранее) экземпляром типа SerializationInfo.

Второй параметр этого специального конструктора является типом StreamingContext, содержащим информацию об источнике или пункте назначения битов. Самым информативным членом этого типа является свойство State, которое представляет значение из перечня StreamingContextStates. Значения этого перечня соответствуют базовой композиции текущего потока.

Честно говоря, если вашей задачей разработки не является низкоуровневый пользовательский сервис удаленного доступа, вам вряд ли придется обращаться к указанному перечню непосредственно. Тем не менее, ниже приводятся имена элементов перечня StreamingContextStates (подробности его описания можно найти в документации .NET Framework 2.0 SDK).

public enum StreamingContextStates {

 Cro*s*sProcess,

 CrossMachine,

 File,

 Persistence,

 Remoting*,*

 Other,

 Clone,

 CrossAppDomain,

 All

}

Чтобы иллюстрировать возможности настройки процесса сериализации с помощью ISerializable, предположим, что у нас есть тип класса, который определяет два элемента строковых данных. Кроме того, предположим, что все символы этих строк должны сохраняться в поток в верхнем регистре, а восстанавливаться из потока – в нижнем. Чтобы учесть эти требования, вы можете реализовать ISerializable так. как показано ниже (не забудьте указать using для пространства имен System.Runtime.Serialization).

[Seriаlizable]

class MyStringData: ISerializable {

 public string dataItemOne, dataItemTwo;

 public MyStringData() {}

 private MyStringData(SerializationInfo si, StreamingContext ctx) {

**// Регидратация члена из потока.**

  dataItemOne = si.GetString(First\_Item").ToLower();

  dataItemTwo = si.GetString("dataItemTwo").ToLower();

 }

 void ISerializable.GetObjectData(SerializatianInfo info, StreamingContext ctx) {

**// Наполнение объекта SerializationInfo**

**// форматированными данными.**

  info.AddValue("First\_Item", dataItemOne.ToUpper());

  info.AddValue("dataItemTwo", dataItemTwo.ToUpper());

 }

}

Обратите внимание на то, что при "наполнении" типа SerializationInfo в методе GetObjectData() *не требуется,*чтобы элементы данных назывались одинаково с внутренними членами-переменными типа. Это может оказаться полезным тогда, когда нужно выделять данные из сохраненного формата. При этом не следует забывать о том, что для получения значений из приватного конструктора необходимо использовать имена, которые назначаются в рамках GetObjectData().

Чтобы проверить пользовательские настройки, предположим, что вы сохранили экземпляр MyStringData с помощью SoapFormatter. Заглянув в результирующий файл \*.soap, вы увидите, что строковые поля в нем действительно представлены в верхнем регистре.

‹SOAP-ENV:Envelope xmlns:xsi="…"›

 ‹SOAP-ENV:Body›

  ‹a1:MyStringData id="ref-1" xmlns:a1="…"›

   ‹First\_Item id="ref-3"›**ЭTO НЕКОТОРЫЕ ДАННЫЕ.**‹/First\_Item›

   ‹dataItemTwo id="ref-4"›**ЭTO НЕКОТОРЫЕ ДОПОЛНИТЕЛЬНЫЕ ДАННЫЕ**‹/dataItemTwo›

  ‹/a1:MyStringData›

 ‹/SOAP-ENV:Body›

‹/SOAP-ENV:Envelope›

**Настройка параметров сериализации с помощью атрибутов**

Хотя реализация интерфейса ISerializable в .NET 2.0 все еще допустима, для настройки процесса сериализации теперь более предпочтительным считается определение методов, наделенных одним из целого ряда новых атрибутов, связанных с задачами сериализации (это атрибуты [OnSerializing], [OnSerialized], [OnDeserializing] и [OnDeserialized]). Использование этих атрибутов оказывается менее громоздким, чем реализация ISerializable, поскольку тогда не возникает необходимости вручную взаимодействовать с поступающим параметром SerializationInfo. Вместо этого вы получаете возможность непосредственно изменять данные состояния во время воздействия форматтера на тип.

При использовании этих атрибутов методы должны определяться так, чтобы они получали параметр StreamingContext и не возвращали ничего (в противном случае в среде выполнения генерируется соответствующее исключение). Обратите внимание на то, что не требуется учитывать все указанные атрибуты сериализации – можно учесть только те стадии сериализации, для которых следует выполнить перехват. Для примера рассмотрите новый тип [Serializable] с теми же требованиями, что и у MyStringData, но на этот раз с использованием атрибутов [OnSerializing] и [OnDeserialized].

[Serializable]

class MoreData {

 public string dataItemOne, dataItemTwo;

**[OnSerializing]**

 internal void OnSerializing(StreamingContext context) {

**// Выполняется в процессе сериализации.**

  dataItemOne = dataItemOne.ToUpper();

  dataItemTwo = dataItemTwo.ToUpper();

 }

**[OnDeserialized]**

 internal void OnDeserialized(StreamingContext, context) {

**// Выполняется по завершении реконструкции объекта.**

  dataItemOne = dataItemOne.ToLower();

  dataItemTwo = dataItemTwo.ToLower();

 }

}

Если выполнись сериализацию этого нового типа, вы снова обнаружите, что данные сохраняются в верхнем регистре, а воcстанавливаются – в нижнем.

**Исходный код**. Проект СustomSerialization размещен в подкаталоге, соответствующем главе 17.

**Поддержка версий сериализации объектов**

В завершение обсуждения этой главы мы рассмотрим тему поддержки версий сериализации объектов. Чтобы понять, почему это необходимо, мы используем следующий сценарий. Предположим, что мы создали класс UserPrefs (он уже упоминался в начале главы) так, как показано ниже.

[Serializable]

class UserPrefs {

 public string objVersion = "1.0";

 public ConsoleColor BackgroundColor;

 public ConsoleColor ForegroundColor;

 public UserPrefs() {

  BackgroundColor = ConsoleColor.Black;

  ForegroundColor = ConsoleColor.Red;

 }

}

Теперь предположим, что у нас есть приложение, в котором выполняется сериализация экземпляра этого класса с помощью BinaryFormatter.

static void Main(string[] args) {

 UserPrefs up*=*new UserPrefs();

 up.BackgroundColor = ConsoleColor.DarkBlue;

 up.ForegroundColor = ConsoleColor.White;

**// Сохранение экземпляра UserPrefs в файле.**

 BinaryFormatter binFormat = new BinaryFormatter();

 Stream fStream = new FileStream(@"C:\user.dat", FileMode.Create, FileAccess.Write, FileShare.None);

 birFormat.Serialize(fStream, up);

 fStream.Сlose();

 Console.ReadLine();

}

К этому моменту экземпляр UserPrefs (версии 1.0) сохранен в C:\user.dat. Но давайте добавим в определение класса UserPrefs два новых поля.

[Serializable]

class UserPrefs {

 public string objVersion = "2.0";

 public ConsoleColor BackgroundColor;

 public ConsoleColor ForegroundColor;

**// Являются новыми!**

 public int BeepFreq;

 public string ConsoleTitle;

 public UserPrefs() {

  BeepFreq = 1000;

  ConsoleTitle = "Моя консоль";

  BackgroundColor = ConsoleColor.Black;

  ForegroundColor = ConsoleColor.Red;

 }

}

Теперь представьте себе, что это же приложение пытается реконструировать экземпляр сохраненного объекта UserPrefs версии 1.0 так, как показано ниже (заметьте, чтобы этот пример работал, предыдущая программная логика сериализации была удалена).

static void Main(string[] args) {

**// Загрузка экземпляра UserPrefs (1.0) в память?**

 UserPrefs up = null;

 BinaryFormatter binFormat = new BinaryFormatter();

 Stream fStream = new FileStream(@"C:\user.dat", FileMode.Open, FileAccess.Read, FileShare.None);

 up = (UserPrefs)binFormat.Deserialize(fStream);

 fStream.Close();

 Console.ReadLine();

}

Вы увидите окно с информацией о следующем исключении, сгенерированном средой выполнения.

Необработанное исключение: System.Runtime.Serialization.SerializationException. Член 'BeepFreq' в классе 'VersionedObject.UserPrefs' не присутствует в сохраненном потоке и не обозначен атрибутом System.Runtime.Serialization.OptionalFieldAttribute.

Проблема в том, что оригинальный объект UserPrefs, сохраненный в C:\user.dat, *не сохранял*два новых поля, присутствующих в обновленном определении класса (это поля BeepFreq и ConsoleTitle). Очевидно, что это настоящая проблема, поскольку для сохраняемого объекта вполне естественно эволюционировать в процессе существования.

До выхода .NET 2.0 единственной возможностью для учета того, что сохраненный объект может не иметь всех новых полей из обновленной и более поздней версии класса, была необходимость реализации ISerializable и осуществление контроля "вручную". Но с появлением .NET 2.0 новые поля могут явно обозначаться атрибутом [Optional Field] (определенным в рамках пространства имен System.Runtime.Serialization).

[Seriаlizable]

class UserPrefs {

 public ConsoleColor BackgroundColor;

 public ConsoleColor ForegroundColor;

**// Являются новыми!**

 [OptionalField]

 public int BeepFreq;

 [OptionalField]

 public string ConsoleTitle;

 public UserPrefs() {

  BeepFreq = 1000;

  ConsoleTitle = ''Моя консоль";

  BackgroundColor = ConsoleColor.Black;

  ForegroundColor = ConsoleColor.Red;

}

Когда форматтер реконструирует объект и обнаруживает, что отсутствующие поля помечены, как необязательные, исключение среды выполнения уже не генерируется. Вместо этого данные, которые были сохранены, проецируется обратно в существующие поля (в данном случае это BackgroundColor и ForegroundColor), a остальным полям присваиваются значения, предусмотренные по умолчанию.

**Замечание**. Следует понимать, что использование [OptionalField] не решает проблему версий сохраненных объектов полностью. Однако этот атрибут обеспечивает решение самой типичной проблемы (добавление новых полей данных). Для решения более сложных задан поддержки версий все же потребуется реализация интерфейса ISerializable.

**Исходный код**. Проект VersionedObject размещен в подкаталоге, соответствующем главе 17.

**Резюме**

В этой главе предлагается обсуждение сервисов сериализации. Вы могли убедиться в том. что платформа .NET для корректного учета всего множества связанных объектов, подлежащих сохранению в потоке, использует объектные графы. Когда каждый член объектного графа обозначен атрибутом [Seriаlizable], данные можно сохранять в любом из нескольких доступных форматов (в двоичном формате, формате SOAP или формате XML).

Вы также узнали о том, что процесс сериализации допускает пользовательскую настройку в рамках двух возможных подходов. Во-первых, у вас есть возможность реализовать интерфейс ISerializable (с поддержкой специального приватного конструктора), чтобы влиять на то. как средства форматирования сохраняют поступающие данные. Во-вторых, вы можете использовать множество новых атрибутов, появившихся в .NET 2.0, которые упрощают процесс сериализации с пользовательскими настройками. Следует просто применить один из атрибутов [OnSerializing], [OnSerialized], [OnDeserializing] или [OnDeserialized] к членам, получающим параметр StreamingContext, и форматтер обработает их соответствующим образом. Завершается глава обсуждением еще одного атрибута, [OptionalField], который может использоваться для поддержки версий сериализации типов.

**ГЛАВА 18. Удаленное взаимодействие .NET**

Разработчики, не имеющие опыта работы с платформой .NET, обычно относят .NET только к средствам создания Интернет-приложений (поскольку ".NET"' часто ассоциируется с "Интернет" и соответствующим программным обеспечением. Вы уже имели возможность убедиться в том, что это далеко не так. Создание Web-приложений является лишь одной и очень узкой (но широко разрекламированной) возможностью платформы .NET. В русле этой информации многие разработчики .NET, не имеющие достаточного опыта, склонны предполагать, что Web-сервисы XML обеспечивают единственный способ взаимодействия с удаленными объектами. Это тоже не соответствует действительности. Используя слой удаленного взаимодействия .NET, можно строить одноранговые распределенные приложения, не имеющие ничего общего с HTTР или XML (если вы этого захотите).

Первой задачей этой главы является рассмотрение низкоуровневых возможностей, используемых средой CLR для передачи информации за границы доменов приложений. При обсуждении проблем удаленного взаимодействия .NET используется множество специальных терминов, таких так агент (т.е. proxy-модуль), канал, маршалинг по ссылке (который противопоставляется маршалингу по значению), серверная активизация объектов (в противоположность клиентской активизации) и т.д… После выяснения сути этих базовых терминов будет предложено несколько примеров программного кода, иллюстрирующих процесс построения распределенных систем в рамках платформы .NET.

**Понятие удаленного взаимодействия .NET**

Вы должны помнить из главы 13, что*домен приложения*[AppDomain] задает логические границы выполнения компоновочного блока .NET в рамках процесса Win32. Понимание этого очень важно для дальнейшего обсуждения распределенных приложений .NET, поскольку *удаленное взаимодействие*означает здесь не более чем взаимодействие двух объектов, сообщающихся через границы доменов. Соответствующие домены приложений могут физически находиться в следующих условиях.

• Два домена приложения определены в рамках одного и того же процесса (и поэтому на одной и той же машине).

• Два домена приложения определены в разных процессах на одной и той же машине.

• Два домена приложения определены в разных процессах на разных машинах.

С учетом этих трех возможностей становится ясно, что удаленное взаимодействие не обязательно предполагает наличие соединенных в сеть компьютеров. На самом деле все примеры, представленные в этой главе, могут вполне успешно выполняться на одной автономной машине. Независимо от расстояния между объектами, в отношении взаимодействующих агентов используются термины "клиент" и "сервер". Упрощенно говоря, *клиент*- это сущность, пытающаяся взаимодействовать с удаленными объектами, а *сервер*- это программный агент, содержащий удаленные объекты.

**Пространства имен удаленного взаимодействия .NET**

Перед тем как углубиться в детали процесса удаленного взаимодействия .NET. мы должны выяснить, какие функциональные возможности предлагают пространства имен, обеспечивающие удаленное взаимодействие. Библиотеки базовых классов .NET содержат очень много пространств имен, позволяющих строить распределенные приложения. Большинство типов, содержащихся в этих пространствах имен, находятся в mscorlib.dll, но дополнения и расширения базовых пространств имен вынесены в отдельный компоновочный блок System.Runtime.Remoting.dll. В табл. 18.1 предлагаются краткие описания пространств имен удаленного взаимодействия .NET 2.0.

**Таблица 18.1.**Пространства имен .NET для поддержки возможностей удаленного взаимодействия

|  |  |
| --- | --- |
| **Пространство имен** | **Описание** |
| System.Runtime.Remoting | Базовое пространство имен, которое должно использоваться при построении любого распределенного приложения .NET |
| System.Runtime.Remoting.Activation | Относительно малое пространство имен, в котором определяются несколько типов, обеспечивающих тонкую настройку процесса активизации удаленного объекта |
| System.Runtime.Remoting.Channels | Содержит типы, представляющие каналы и приемники каналов |
| Systern.Runtime.Remoting.Channels.Http | Содержит типы, использующие протокол HTTP для транспорта сообщений и объектов в удаленную точку и обратно |
| System.Runtime.Remoting.Channels.Ipc | Пространство имен, которое появилось в .NET 2.0 и содержит типы, использующие архитектуру IPC Win32. Архитектура IPC (Interprocess Communication – взаимодействие процессов) обеспечивает быстрое взаимодействие доменов приложений, существующих на одной физической машине |
| System.Runtime.Remoting | Базовое пространство имен, которое должно использоваться при построении любого распределенного приложения .NET |
| System.Runtime.Remoting.Activation | Относительно малое пространство имен, в котором определяются несколько типов, обеспечивающих тонкую настройку процесса активизации удаленного объекта |
| System.Runtime.Remoting.Channels | Содержит типы, представляющие каналы и приемники каналов |
| System.Runtime.Remoting.Channels.Http | Содержит типы, использующие протокол HTTP для транспорта сообщений и объектов в удаленную точку и обратно |
| System.Runtime.Remoting.Channels.Ipc | Пространство имен, которое появилось в .NET 2.0 и содержит типы, использующие архитектуру IPC Win32. Архитектура IPC (Interprocess Communication – взаимодействие процессов) обеспечивает быстрое взаимодействий доменов приложений, существующих на одной физической машине |
| System.Runtime.Remoting.Channels.Tcp | Содержит типы, использующие протокол TCP для транспорта сообщений и объектов в удаленную точку и обратно |
| System.Runtime.Remoting.Contexts | Позволяет конфигурировать параметры объектного контекста |
| System.Runtime.Remoting.Lifetime | Содержит типы, управляющие циклом существования удаленных объектов |
| System.Runtime.Remoting.Messaging | Содержит типы, используемые для создания и передачи объектов сообщений |
| System.Runtime.Remoting.Metadata | Содержит типы, используемые для настройки параметров генерированиям форматирования сообщений SOAP |
| System.Runtime.Remoting.Metadata.W3cXsd2001 | Содержит типы, представляющие формат XSD (XML Schema Definition – определение схемы XML) в соответствии со стандартами Консорциума W3C, принятыми в 2001 году |
| System.Runtime.Remoting.MetadataServices | Содержит типы, используемые средством командной строки soapsuds.exe при конвертировании метаданных удаленной инфраструктуры .NET в XML-схемы (и обратно) |
| System.Runtime.Remoting.Proxies | Содержит типы, обеспечивающие функциональные возможности для объектов, выполняющих задачи агента (proxy) |
| System.Runtime.Remoting.Services | Определяет ряд общих базовых классов (и интерфейсов), которые обычно используются только внутренними агентами удаленного взаимодействия |

**Каркас удаленного взаимодействия .NET**

Когда клиенты и серверы обмениваются информацией через границы приложений, среда CLR вынуждена использовать низкоуровневые примитивы, обеспечивающие настолько "прозрачное" взаимодействие сторон, насколько это возможно. Это значит, что вам, как программисту .NET, не нужно создавать огромные по объему блоки программного кода поддержки сетевого соединения, чтобы вызвать метод удаленного объекта. Также и серверному процессу *не нужно*"вручную" извлекать сетевой пакет из очереди и преобразовывать сообщение в формат, понятный удаленному объекту. Вы вправе ожидать, что среда CLR позаботится о таких деталях сама, используя свой стандартный набор примитивов удаленного взаимодействия (хотя, при желании, вы тоже можете принять участие в установке параметров соответствующего процесса).

В сущности, слой удаленного взаимодействия .NET обеспечивает аккуратную совместную работу следующих четырех ключевых элементов:

• агенты;

• сообщения;

• каналы;

• форматтеры.

Давайте рассмотрим каждый из указанных элементов по очереди и выясним, как их комбинация позволяет осуществлять удаленные вызовы методов.

**Агенты и сообщения**

Клиенты и объекты сервера взаимодействуют не напрямую, а через посредника, обычно называемого агентом (или proxy-модулем). Роль агента .NET заключается в создании для клиента иллюзии того, что он взаимодействует с запрошенным удаленным объектом *в одном домене приложения.*Чтобы создать такую иллюзию, агент предлагает интерфейс (члены, свойства, поля и т.д.), идентичный интерфейсу удаленного типа. С точки зрения клиента данный агент и *является*удаленным объектом. Однако "за кулисами" агент переправляет вызовы удаленному объекту.

Формально такой агент, вызываемый клиентом непосредственно, называется *прозрачным агентом*(transparent proxy). Этот объект, генерируемый средой CLR автоматически, несет ответственность за проверку того, что при вызове удаленного метода клиент получит нужное число параметров (и они будут нужного типа). Поэтому прозрачный агент можно интерпретировать, как фиксированный слой взаимодействия, который нельзя программно изменить или расширить.

В предположении о том, что прозрачный агент может выполнять проверку входных аргументов, соответствующая информация упаковывается в другой генерируемый средой CLR тип, который называется*объектом сообщения.*По определению все объекты сообщений реализуют интерфейс System.Runtime.Remoting.Messaging.IMessage.

**public interface IMessage**{

 IDictionary Properties { get; }

}

Как видите, интерфейс IMessage определяет единственное свойство (с именем Properties), которое обеспечивает доступ к коллекции, используемой для хранения предоставленных клиентом аргументов. После наполнения объекта сообщения содержимым средой CLR, он будет передан родственному типу, называемому *реальным агентом*(real proxy).

Реальный, агент – это сущность, которая фактически посылает объект сообщения в канал (понятие канала будет обсуждаться ниже). Реальный агент, который (в отличие от прозрачного агента) *может быть расширен*программистом, представляется базовым типом класса с именем RealProxy (что и следовало ожидать). Снова следует подчеркнуть, что среда CLR генерирует клиентскую реализацию реального агента для использования по умолчанию, которая вполне подойдет вам если не во всех, то в большинстве случаев. Но чтобы иметь представление о функциональных возможностях, предлагаемых абстрактным базовым классом RealProxy, изучите формальное определение этого типа.

public abstract class RealProxy: object {

 public virtual ObjRef CreateObjRef(Type requestedType);

 publiс virtual bool Equals(object obj);

 public virtual IntPtr GetCOMIUnknown(bool fIsMarshalled);

 public virtual int GetHashCode();

 public virtual void GetObjectData(SerializationInfo info, StreamingContext context);

 public Type GetProxiedType();

 public static object GetStubData(RеаlРrоxу rp);

 public virtual object GetTransparentProxy();

 public Type GetType();

 public IConstructionReturnMessage InitializeServerObject(IConstructionCallMessage ctorMsg);

 public virtual IMessage **Invoke**(IMessage msg);

 public virtual void SetCOMIUnknown(IntPtr i);

 public static void SetStubData(RealProxy rp, object stubData);

 public virtual IntPtr SupportsInterface(ref Guid iid);

 public virtual string ToString();

}

Если вы не заняты построением пользовательской реализации реального агента клиента, то единственным интересным для вас членом будет RealProxy.Invoke(). С помощью метода Invoke() сгенерированный средой CLR прозрачный агент в фоновом режиме передает форматированный объект сообщения типу RealProxy.

**Каналы**

После того как агенты проверят и отформатируют поставляемые клиентом аргументы, упаковав их в объект сообщении, соответствующий IMessage-совместимый тип передается от реального агента объекту канала. Каналы – это сущности, отвечающие за транспортировку сообщения удаленному объекту и, если это необходимо, за то, чтобы возвращаемое значение от удаленного объекта было доставлено обратно клиенту. В библиотеках базовых классов .NET 2.0 предлагаются готовые реализации трех каналов:

• TCP-канал;

• HTTP-канал;

• IPC-канал.

TCP-канал представляется типом класса TcpChannel и используется для передачи сообщений с использованием сетевого протокола TCP/IP. Класс TcpChannel удобен тем, что форматированные пакеты оказываются исключительно "легкими", поскольку сообщения превращаются в плотный двоичный формат с помощью BinaryFormatter (да, именно того BinaryFormatter, о котором шла речь в главе 17). При использовании типа TcpChannel удаленный доступ осуществляется быстрее. Недостатком является то, что TCP-каналы не согласуются с брандмауэром автоматически и могут требовать вмешательства сервисов администратора системы, чтобы получить разрешение на пересечение границы машины.

В противоположность этому, *HTTP-канал,*представляемый типом класса HttpChannel, преобразует объекты сообщений в формат SOAP, используя для этого соответствующий форматтер SOAP. Выше вы могли убедиться в том, что SOAP опирается на XML и поэтому результат в данном случае оказывается более объемным, чем в случае TcpChannel. Поэтому при использовании HttpChannel удаленный доступ может осуществляться медленнее. Но, с другой стороны, протокол HTTP является гораздо более дружественным в отношении брандмауэра, поскольку большинство сетевых экранов позволяет текстовым пакетам направляться через порт с номером 80.

Наконец, в .NET 2.0 предлагается доступ к *IPC-каналу,*представленному типом IpcChannel, который определяет коммуникационный канал связи для удаленного взаимодействия с использованием IPC-архитектуры операционной системы Windows. Ввиду того, что IpcChannel при пересечении доменов приложений действует в обход традиционных систем сетевой коммуникации, IpcChannel оказывается намного быстрее, чем HTTP- и TCP-каналы, однако, может использоваться только для взаимодействия доменов приложения на *одном и том же компьютере.*Поэтому IpcChannel *не*может применяться для построения распределенных приложений, допускающих использование множества физических компьютеров. Но тип IpcChannel может оказаться идеальным вариантом тогда, когда вы хотите обеспечить наивысшую скорость обмена информацией между двумя локальными программами.

Важно понимать, что вне зависимости от типа канала, который вы выберете для использования, и HttpChannel, и TcpChannel, и IpcChannel реализуют интерфейсы IChannel, IChannelSender и IChannelReceiver. Интерфейс IChannel (как вы вскоре убедитесь) определяет небольшой набор членов, обеспечивающих общую функциональность всех типов каналов. Роль IChannelSender заключается в определении для каналов общего множества членов, позволяющих отправлять информацию данному получателю. С другой стороны, IChannelReceiver определяет множество членов, позволяющих каналу*получать*информацию данного отправителя.

Чтобы позволить приложениям клиента и сервера зарегистрировать выбранный ими канал, вы должны использовать метод ChannelServices.RegisterChannel(), который получит тип, реализующий IChannel. Вот фрагмент программного кода, который показывает, как домен серверного приложения может зарегистрировать HTTP-канал, использующий порт 32469 (аналогичные возможности клиента будут продемонстрированы чуть позже).

**// Создание и регистрация HttpChannel-сервера с портом 32469.**

HttpChannel c = new HttpChannel(32469);

ChannelServices.RegisterChannel(с);

**Снова о роли форматтера .NET**

Заключительным элементом головоломки удаленного взаимодействия .NET является форматтер. Типы TcpChannel и HttpChannel используют свои внутренние форматтеры, задачей которых является перевод объекта сообщения в термины соответствующего протокола. Как уже говорилось, тип TcpChannel использует тип BinaryFormatter, в то время как тип HttpChannel использует функциональные возможности типа SoapFormatter. Опираясь на знания, полученные в предыдущей главе, вы должны понимать, как соответствующий канал форматирует поступающие сообщения.

После создания форматированного сообщения оно передается в канал, по которому в конце концов достигает целевого домена приложения. Там это сообщение преобразуется из специфических терминов протокола обратно в термины .NET, после чего элемент, который называется *диспетчер,*вызывает нужный метод удаленного объекта.

**Общая картина**

Если у вас от чтения предыдущих разделов уже голова идет кругом, не паникуйте! Прозрачный агент, реальный агент, объект сообщения и диспетчер вы можете, как правило, просто игнорировать, поскольку чаще всего вам вполне подойдут параметры удаленного взаимодействия, предлагаемые по умолчанию. Чтобы закрепить в памяти соответствующую последовательность событий, рассмотрите рис. 18.1, на котором показана схема процесса коммуникации двух объектов из разных доменов приложений.

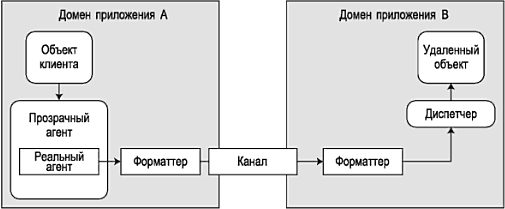


Рис. 18.1. Архитектура удаленного взаимодействия .NET, предлагаемая по умолчанию

**Несколько слов о расширении стандартных возможностей**

Ключевой особенностью слоя удаленного взаимодействия .NET является то, что большинство предлагаемых по умолчанию слоев удаленного взаимодействия может быть расширено или полностью заменено разработчиком приложения. Так, если вы хотите (или, возможно, вам нужно) построить диспетчер пользовательских сообщений, пользовательский форматтер или реальный агент, вы имеете для этого все возможности. Вы также можете добавить *дополнительные*слои, включив в цепочку обработки пользовательские типы (например, пользовательский приемник, используемый для предварительной или последующей обработки сообщений). Вам лично, возможно, никогда и не придется модифицировать базовый слой удаленного взаимодействия .NET, но факт в том, что платформа .NET предлагает пространства имен, позволяющие решить такую задачу.

**Замечание.**В этой главе тема расширения базового слоя удаленного взаимодействия .NET не обсуждается. Чтобы узнать, как это сделать, обратитесь к книге Ingo Rammer, *Advanced .NET Remoting*(Apress, 2002).

**Термины удаленного взаимодействия .NET**

Подобно любой новой парадигме, удаленное взаимодействие .NET предлагает свой собственный набор трехбуквенных акронимов. Поэтому, перед тем как рассмотреть первый пример программного кода, нам с вами придется определить несколько терминов, обычно используемых при описании приложения удаленного взаимодействия .NET. Как вы можете догадаться сами, соответствующая терминология используется для описания ответов на ряд общих вопросов, возникающих при построении распределенного приложения. Как передать тип через границы домена приложения? Когда именно будет активизирован удаленный тип? Как управлять циклом существования удаленного объекта (и т.д.)? Когда вы поймете соответствующую терминологию, вопросы построения распределенных приложений .NET уже не будут вам казаться столь запутанными.

**Варианты маршалинга для объектов: MBR и MBV**

В рамках платформы .NET вы имеете на выбор два варианта того, как предоставить удаленный объект клиенту. Упрощенно говоря, *маршалинг*описывает правила передачи удаленного объекта из одного домена приложения в другой. При разработке объекта, предусматривающего удаленное использование, вы можете выбрать либо семантику *MBR*(marshal-by-reference – маршалинг по ссылке), либо семантику*MBV*(marshal-by-value – маршалинг по значению). Их различие заключается в следующем.

• *MBR-объекты.*Вызывающая сторона получает агента для осуществления доступа к удаленному объекту.

• *MBV-объекты.*Вызывающая сторона получает полную копию объекта для использования в своем домене приложения.

При использовании типа, относящегося к MBR-объектам, среда CLR обеспечит создание в домене приложения клиента прозрачного и реального агентов, в то время как сам MBR-объект будет оставаться в домене приложения сервера. При вызове методов удаленного типа клиентом система удаленного взаимодействия .NET (схема которой описана выше) активизируется, чтобы выполнить задачи упаковки, отправки и получения информации при обмене данными через границы доменов приложений. Для этого MBR-объекты имеют ряд свойств, "простирающихся" за рамки их физического расположения. Вы увидите, что MBR-объекты имеют различные опции конфигурации, относящиеся к их активизации и управлению циклом существования. В противоположность этому, MBV-объекты представляют собой локальные *копии*удалённых объектов (использующие протокол сериализации .NET, который был рассмотрен в главе 17). MBV-объекты имеют намного меньше опций конфигурации, поскольку их цикл существования контролируется непосредственно клиентом. Подобно любому другому объекту .NET, после того как клиент освободит все ссылки на MBV-тип, этот тип становится потенциальным объектом внимания для сборщика мусора. Поскольку MBV-типы являются локальными копиями удаленных объектов, процесс вызова клиентом членов соответствующего типа, вообще говоря, не предполагает никакой сетевой активности.

Следует понимать, что вполне естественным для сервера является поддержка доступа к множеству MBR- и MBV-типов. Вы можете также догадаться, что MBR-типы обычно поддерживают методы, возвращающие различные MBV-типы, что, в общем-то, напоминает автоматизированное предприятие, где один объект создает и выпускает другие связанные объекты. Здесь возникает следующий вопрос: как сконфигурировать пользовательский тип класса для использования в виде MBR-или MBV-объекта?

**Конфигурация MBV-объекта**

Процесс конфигураций объекта для использования в виде MBV-типа абсолютно аналогичен процессу конфигурации объекта для сериализации. Просто объявите соответствующий тип с атрибутом [Serializable].

**[Serializable]**

public class SportsCar {…}

**Конфигурация MBR-объекта**

MBR-объекты не маркируются специальным атрибутом .NET, а получаются (явно или неявно) из базового класса System.MarshalByRefObject.

public class SportsCarFactory: **MarshalByRefObject**{…}

Формально тип MarshalByRefObject определяется следующим образом.

public abstract class **MarshalByRefObject:** object {

 public virtual ObjRef CreateObjRef(Type requestedType);

 public virtual bool Equals(object obj);

 public virtual int GetHashCode();

 public virtual object GetLifetimeService();

 public Type GetType();

 public virtual object InitializeLifetimeService();

 public virtual string ToString();

}

Функциональные возможности, наследуемые от System.Object, вполне понятны, а роль остальных членов описана в табл. 18.2.

**Таблица 18.2.**Основные члены System.MarshalByRefObject

|  |  |
| --- | --- |
| **Член** | **Описание** |
| CreateObjRef() | Создает объект, содержащий всю информацию, необходимую для генерирования агента, который будет использоваться для взаимодействия с удаленным объектом |
| GetLifetimeServices() | Возвращает текущий сервис-объект, контролирующий политику цикла существования для данного экземпляра |
| InitializeLifetimeServices() | Генерирует сервис-объект для контроля политики цикла существования данного экземпляра |

Можно сказать, что суть типа MarshalByRefObject заключается в определении членов, которые затем могут переопределяться для того, чтобы программно управлять циклом существования MBR-объекта (подробнее об управлении циклом существования объектов будет говориться в этой главе позже).

**Замечание.**То, что вы сконфигурировали тип в виде MBV- или MBR-объекта, совсем не означает, что этот объект следует использовать только в приложении удаленного взаимодействия, а означает только то, что этот объект *можно*использовать в таком приложении. Например, тип System.Windows.Forms.Form является потомком MarshalByRefObject. Поэтому при удаленном доступе он реализуется как MBR-тип, а в других случаях он будет обычным локальным объектом в домене приложения клиента.

**Замечание.**Как следствие предыдущего замечания обратим внимание на то, что если тип .NET не предполагает сериализацию и в его цепочке наследования нет MarshalByRefObject, то такой тип может активизироваться и использоваться только в его исходном домене приложения, т.е, такой тип является контекстно-связанным (см. главу 13).

Теперь, когда вы четко понимаете суть различий между MBR- и MBV-типами, давайте рассмотрим некоторые проблемы, специфичные для MBR-типов (к MBV-типам это не относится).

**Варианты активизации для MBR-типа: WKO и CAO**

Еще одной проблемой выбора, возникающей перед вами, как программистом, является принятие решения о том, *когда*следует активизировать MBR-объект и *когда*этот объект должен стать кандидатом для участия в процедуре сборки мусора на сервере. На первый взгляд, такая постановка вопроса может показаться странной, поскольку, очевидно. MBR-объекты должны создаваться тогда, когда клиент их запрашивает, а уничтожаться тогда, когда клиент заканчивает работать с ними. Конечно, именно клиент предоставляет слою удаленного взаимодействия информацию о своем желания взаимодействовать с удаленным типом, но в ответ на запрос клиента серверное приложение имеет возможность создать соответствующий тип не сразу.

Причина такого, казалось бы. странного поведение связана с оптимизацией. Точнее, каждый MBR-тип можно настроить на активизацию с использованием одного из двух следующих подходов:

• как общеизвестный объект (Well-Known Object – WKO);

• как объект, активируемый клиентом (Client Activated Object – CAO).

Замечание. Потенциальным источником недоразумений здесь является то, что в литературе, посвященной .NET, вместо акронима WKO также используют SAO (Server Activated Object – объект, активизируемый сервером). Акроним SAO встречается в целом ряде статей и книг, связанных с .NET. В этой главе, в соответствии с современной терминологией, будет использоваться аббревиатура WKO.

WKO-объекты – это MBR-типы, цикл существования которых подконтролен непосредственно домену приложения сервера. Приложение клиента активизирует удаленный тип, используя понятное общеизвестное строковое имя (отсюда и возник термин WKO). Домен приложения сервера размещает WKO-типы тогда, когда клиент выполняет первый вызов метода данного объекта (через прозрачный агент), а *не тогда,*когда программный код клиента использует ключевое слово new или когда вызов происходит через статический метод Activator.GetObject(), например:

**// Получение агента для удаленного объекта.**

**// Эта строка не приводит к немедленному создании WKO-типа!**

object remoteObj = Activator.GetObject(/\* параметры… \*/);

**// Вызов метода удаленного WKO-типа. Это приводит к созданию**

**// WKO-объекта и вызову метода ReturnMessage**().

RemoteMessageObject simple = (RemoteMessageObject)remoteObj;

Console. WriteLine("Сервер отвечает: {0}", simple.ReturnMуssage());

В чем здесь здравый смысл? При таком подходе простое предложение создать объект не ведет к немедленному пику сетевого обмена данными. Другим следствием является то, что WKO-типы могут создаваться *только с помощью конструктора, заданного по умолчанию.*Это разумно, поскольку конструктор удаленного типа используется только тогда, когда клиент выполняет вызов члена. Так что среда выполнения не имеет никакого иного варианта выбора, кроме вызова конструктора, заданного типом по умолчанию.

Замечание. Всегда помните о том, что любой WKD-тип должен иметь конструктор, заданный по умолчанию!

Если вы хотите разрешить клиенту создавать удаленные MBR-объекты с помощью пользовательского конструктора, сервер должен сконфигурировать соответствующий объект, как САО-объект. Цикл существования САО-объектов контролируется доменом приложения клиента. При доступе к САО-типу соответствующий обмен данными с сервером происходит уже при использовании клиентом ключевого слова new (с любым конструктором типа) или типа Activator.

**Варианты конфигурации WKO-типа: синглеты и объекты одиночного вызова**

Наконец, еще одна проблема выбора для MBR-типов в проекте .NET связана с тем, как сервер должен обрабатывать множественные обращения к WKO-типу. С САО-типами эта проблема не возникает, поскольку для них всегда есть однозначное соответствие между клиентом и удаленным САО-типом (эти типы являются объектами, кумулятивно изменяющими параметры своего состояния в процессе выполнения вызовов клиентов).

Одним из вариантов является конфигурация WKO-типа в виде *синглета.*В этом случае среда CLR создаст один экземпляр удаленного типа, который будет принимать запросы любого числа клиентов. Этот вариант оказывается естественным тогда, когда нужно поддерживать состояние типа, одинаковое для всех абонентов, выполняющих удаленные вызовы. Множество клиентов могут вызывать один и тот же метод в одно и то же время, поэтому среда CLR помещает каждый вызов клиента в новый поток. Однако обеспечение гарантий того, что ваши объекты будут реентерабельны, является *вашей*обязанностью, и для этого следует использовать подходы, описанные в главе 14,

В противоположность синглету, *объект одиночного вызова*- это WKO-тип, существующий только в контексте вызова отдельного метода. Поэтому, например, если WKO-тип, сконфигурированный с учетом семантики одиночного вызова, используется 20 клиентами, то сервер создаст 20 отдельных объектов (по одному для каждого клиента), и все эти объекты станут кандидатами для участия в процессе сборки мусора сразу же после завершения вызова метода. Как вы можете догадаться, объекты одиночного вызова, будучи объектами, не меняющими своего состояния, поддаются масштабированию лучше, чем синглеты.

Задача определения конфигурации состояния WKO-типа возлагается на сервер. Программно указанные варианты задаются с помощью перечня System.Runtime.Remoting.WellKnownObjectMode.

public enum **WellKnownObjectMode**{

 SingleCall,

 Singleton

}

**Сводная характеристика MBR-объектов**

Вы имели возможность убедиться в том, что для конфигурации MBV-объек-тов долгих размышлений не потребуется: нужно просто применить атрибут [Serializable], чтобы позволить отправку копий соответствующего типа в домен приложения клиента. С этого момента все взаимодействие с MBV-типом происходит в локальном окружении клиента. Когда клиент завершит использование соответствующего MBV-типа, этот тип становится объектом внимания сборщика мусора, и никаких проблем не возникает.

Но для MBR-типов имеется целый ряд вариантов конфигурации. Вы видели, что MBR-тип допускает варианты конфигурации в отношении его времени активизации, состояния и управления циклом существования. Чтобы представить весь набор имеющихся возможностей, в табл. 18.3 показано, как WKO и САО-объекты соотносятся с вариантами поведения, которые только что были нами рассмотрены.

**Таблица 18.3.**Опции конфигурации для MBR-типов

|  |  |  |
| --- | --- | --- |
| **Характеристика MBR-объекта** | **Поведение WKO-типа** | **Поведение САО-типа** |
| Опции создания экземпляра | WKO-типы могут активизироваться только c помощью конструктора, заданного по умолчанию, который запускается при первом вызове метода клиентом | CAO-типы могут активизироваться с помощью любого конструктора типа. Удаленный объект создается тогда, когда вызывающая сторона использует семантику конструктора (или тип Activate) |
| Управление состоянием | WKO-типы можно сконфигурировать, как синглет или объект одиночного вызова. Синглет может обслуживать множество клиентов и является объектом, кумулятивно изменяющим параметры своего состояния в процессе выполнения вызовов клиентов. Объект одиночного вызова существует только в процессе данного вызова клиента и является объектом, не меняющим своего состояния в процессе выполнения | Цикл существования САО-типа контролируется вызывающей стороной, поэтому САО-типы являются объектами, кумулятивно изменяющими параметры своего состояния в процессе выполнения вызовов клиентов |
| Управление циклом существования | Для WKO-типов, являющихся синглетами, используется схема лизингового управления (которая будет описана в этой главе позже). WKO-типы, являющиеся объектами одиночного вызова, оказываются объектами внимания для сборщика мусора сразу же по завершении вызова метода | Для CAO-типов используется схема лизингового управления (которая будет описана в этой главе позже) |

**Инсталляция приложения, использующего удаленное взаимодействие**

Хватит акронимов! К этому моменту вы почти готовы к построению своего первого .NET-приложения, использующего удаленное взаимодействие. Но перед тем, как это сделать, мы должны обсудить одну деталь: процедуру инсталляции. При создании приложения удаленного взаимодействия .NET вы, скорее всего, будете иметь три (да, именно три, а не два) разных компоновочных блока .NET, составляющих ваше приложение. Я уверен, что первые два компоновочных блока вы смо-жете указать сами.

• *Клиент*. Этот компоновочный блок представляет сущность (например, приложение Windows Forms или консольное приложение), заинтересованную в получении доступа к удаленному объекту.

• *Сервер.*Этот компоновочный: блок представляет сущность, получающую канальные запросы от удаленного клиента и обслуживающую удаленные объекты.

Но к чему же тогда отнести третий компоновочный блок? Во многих случаях приложение сервера обслуживает третий компоновочный блок, определяющий и реализующий удаленные объекты. Для удобства я буду называть этот компоновочный блок *общим компоновочным блоком.*Такое разделение компоновочного блока, содержащего удаленные объекты, и хоста сервера оказывается очень важным, поскольку компоновочные блоки и клиента, и сервера устанавливают ссылки на общий компоновочный блок, чтобы получить метаданные типов, допускающих удаленный доступ.

В простейшем случае общий компоновочный блок размещается в каталогах приложений и клиента, и сервера. Потенциальным недостатком такого подхода является то, что клиент ссылается на компоновочный блок, содержащий программный CIL-код, который никогда не используется (и соответствующий программный код нельзя будет скрыть от конечного пользователя). Чтобы устранить этот недостаток, заметим, что общий компоновочный блок нужен клиенту только для получения метаданных удаленных типов. Но это можно обеспечить и следующими способами.

• Сконструировать удаленные объекты с применением программных технологий, использующих интерфейсы. В этом случае клиент может установить ссылку на двоичный блок .NET, содержащий только определения соответствующих интерфейсов, и ничего более.

• Использовать приложение командной строки soapsuds.exe. С помощью этого инструмента можно сгенерировать компоновочный блок, содержащий только метаданные удаленных типов.

• Вручную построить компоновочный блок, содержащий только метаданные удаленных типов.

Тем не менее, чтобы упростить изложение материала этой главы, мы с вами построим и установим общие компоновочные блоки, содержащие как необходимые метаданные, так и CIL-код реализации.

**Замечание.**Чтобы выяснить, как реализовать общие компоновочные блоки в рамках указанных выше альтернативных подходов, прочитайте книгу Tom Barnaby, *Distributed .NET Programming in*C# (Apress, 2002).

**Создание распределенного приложения**

Ничто не принесет большей радости, чем создание реального распределенного приложения на новой платформе. Чтобы показать, как быстро можно создать и запустить приложение, использующее слой удаленного взаимодействия .NET, мы построим простой пример такого приложения. Выше уже отмечалось, что такое приложение должно состоять из трех компоновочных блоков .NET:

• общий компоновочный блок с именем SimpleRemotingAsm.dll;

• компоновочный блок клиента с именем SimpleRemoteObjectClient.exe;

• компоновочный блок сервера с именем SimpleRemoteObjectServer.exe.

**Создание общего компоновочного блока**

Сначала создадим общий компоновочный блок, SimpleRemotingAsm.dll, на который будут ссылаться как сервер, так и клиент. В SimpleRemotingAsm.dll определяется единственный MBR-тип с именем RemoteMessageObject, который поддерживает два открытых члена. Метод DisplayMessage() выводит в окно консоли сервера поставляемое клиентом сообщение, a ReturnMessage() возвращает некоторое сообщение клиенту. Вот полный программный код этой новой библиотеки классов C#.

namespace SimpleRemotingAsm {

**// Для этого типа при удаленном доступе**

**// будет иcпользоваться маршалинг до ссылке (MBR).**

 public class RemoteMessageObject: **MarshalByRefObject**{

  public RemoteMessageObject() { Console.WriteLine("Создание RemoteMessageObject!"); }

**// Этот метод получает входную строку**

**// от вызывающей стороны.**

  public void DisplayMessage(string msg) { Console.WriteLine("Сообщение: {0}", msg); }

**// Этот метод возвращает значение вызывающей стороне.**

  public string ReturnMessage() { return "Привет от сервера!"; }

 }

}

Наиболее интересным здесь является то, что соответствующий тип получается из базового класса System.MarshalByRefObject, в результате чего полученный класс будет гарантированно доступным с помощью агента на стороне клиента. Также обратите внимание на пользовательский вариант конструктора, заданного по умолчанию, который печатает сообщение при создании экземпляра типа. Вот и все. Теперь можете создать новый компоновочный блок SimpleRemotingAsm.dll на базе этого программного кода.

**Создание компоновочного блока сервера**

Напомним, что компоновочные блоки сервера обслуживают, в частности, и общие компоновочные блоки, содержащие объекты удаленного доступа. Создайте консольную программу с именем SimpleRemoteObjectServer. Роль серверного компоновочного блока заключается в том, чтобы открыть канал для поступающих запросов и зарегистрировать RemoteMessageObjесt, как WKO-объект. Сначала сошлитесь на компоновочные блоки System.Runtime.Remoting.dll и SimpleRemotingAsm.dll и обновите Main() так, как предлагается ниже.

using System;

using System.Runtime.Remoting;

using System.Runtime.Remoting.Channels;

using System.Runtime.Remoting.Channels.Http;

**using SimpleRemotingAsm;**

namespace SimpleRemoteObjectServer {

 class SimpleObjServer {

  static void Main(string[] args) {

   Console.WriteLine("\*\*\* Начало работы SimpleRemoteObjectServer! \*\*\*");

   Console.WriteLine("Для завершения нажмите ‹Enter›");

**// Регистрация нового HttpChannel**

   HttpChannel с = new HttpChannel(32469);

   ChannelServices.RegisterChannel(c);

**// Регистрация WKO-типа с активацией синглета.**

   RemotingConfiguration.RegisterWellKnownServiceType(typeof(SimpleRemotingAsm.RemoteMessageObject), "RemoteMsgObj.soap", WellKnownObjectMode.Singleton);

   Console.ReadLine();

  }

 }

}

Метод Main() начинается c создания нового типа HttpChannel, для которого указан произвольный идентификатор порта. Этот порт открывается путем регистрации канала с помощью статического метода ChannelServices.RegisterChannel(). После регистрации канала компоновочный блок удаленного сервера может обрабатывать сообщения, поступающие через порт с номером 32469.

**Замечание.**Номер, который вы назначите порту, как правило, выбираете вы сами (или ваш системный администратор). При этом, однако, следует учитывать то, что порты с номерами ниже 1024 резервируются для использования системой.

Затем, чтобы зарегистрировать тип SimpleRemotingAsm.RemoteMessageObject в качестве WKO-типа, используется метод RemotingConfiguration.RegisterWellKnownServiceType(). Первым аргументом этого метода является информация типа для регистрируемого типа. Вторым параметром RegisterWellKnownServiceТуре() является произвольная выбранная вами строка, которая будет использоваться для идентификации регистрируемого объекта при обмене данными между доменами приложений. Здесь вы информируете среду CLR о том, что данный объект должен распознаваться клиентом по имени RemoteMsgObj.soap.

Заключительным параметром является член перечня WellKnownObjectMode, и для него здесь указано WellKnownObjectMode.Singleton. Напомним, что при использовании WKO-синглета все поступающие запросы обслуживаются одним экземпляром RemoteMessageObject. Создайте компоновочный блок сервера и переходите к созданию программного кода клиента.

**Создание компоновочного блока клиента**

Теперь, когда у вас есть приемник, который будет обслуживать объекты уда-ленного доступа, остается создать компоновочный блок, который запросит доступ к соответствующим возможностям. Здесь снова создайте простое консольное приложение. Установите ссылку на System.Runtime.Remoting.dll и SimpleRemotingAsm.dll. Реализуйте Main() так, как показано ниже.

using System;

using System.Runtime.Remoting;

using System.Runtime.Remoting.Channels;

using System.Runtime.Remoting.Channels.Http;

**using SimpleRemotingAsm;**

namespace SimpleRemoteObjectClient {

 class SimpleObjClient {

  static void Main(string[] args) {

   Console.WriteLine("\*\*\* Начало работы SimpleRemoteObjectClient! \*\*\*");

   Console.WriteLine("Для завершения нажмите ‹Enter›");

**// Создание нового HttpChannel.**

   HttpChannel с = new HttpChannel();

   ChannelServices.RegisterChannel(c);

**// Получение агента для удаленного доступа к WKO-типу.**

   object remoteObj = Activator.GetObject(typeof(SimpleRemotingAsm.RemoteMessageObject), "http://localhost:32469/RemoteMsgObj.soap");

**// Использование удаленного объекта.**

   RemoteMessageObject simple = (RemoteMessageObject)remoteObj;

   simple.DisplayMessage("Привет от клиента!");

   Console.WriteLine("Сервер говорит: {0}", simple.ReturnMessage());

   Console.ReadLine();

  }

 }

}

В этом приложении клиента обратите внимание на следующее. Во-первых, клиент также должен зарегистрировать HTTP-канал, но идентификатор порта при этом не указывается, поскольку конечная точка канала задается адресом URL активизации, поставляемым клиентом. Поскольку клиент взаимодействует с WKO-типом, вы должны активизировать конструктор типа, заданный по умолчанию. С этой целью вызывается метод Activator.GetObject() с двумя параметрами. Первым параметром является информация типа удаленного объекта, с которым вы хотите взаимодействовать. Прочитайте последнее предложение еще раз. Поскольку здесь метод Activator.GetObject() требует метаданные описания объекта, становится ясно, почему для клиента также требуется ссылка на общий компоновочный блок! В конце главы будут рассмотрены различные возможности совершенствования поведения компоновочного блока клиента в этом отношении.

Второй параметр метода Activator.GetObject() представляет собой URL *активизации.*Значение URL активизации, описывающее WKO-тип, можно представить в следующем обобщенном формате.

СхемаПротокола://ИмяКомпьютера:Порт/UriОбъекта

Наконец, заметим, что метод Activator.GetObject() возвращает общий тип System.Object, поэтому для получения доступа к членам RemoteMessageObject необходимо использовать явное преобразование типа.

**Тестирование приложения, использующего удаленное взаимодействие**

При тестировании приложения начните с запуска серверного приложения, которое откроет HTTP-канал и зарегистрирует объект RemoteMessageObject для удаленного доступа. Затем запустите экземпляр приложения клиента. Если все пройдет хорошо, окно вашего сервера должно иметь вид, показанный на рис. 18.2, а приложение клиента должно отображать то, что вы видите на рис. 18.3.



Рис. 18.2. Вывод сервера

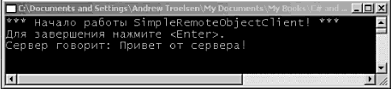


Рис. 18.3. Вывод клиента

**Тип ChannelServices**

Итак, объявляя существование удаленного типа, сервер использует тип System. Runtime.Remoting.Channels.ChannelServices. Тип ChannelServices предлагает небольшой набор статических методов, призванных обеспечить содействие в процессе регистрации канала удаленного взаимодействия и обнаружения указанного URL. Главные члены данного типа описаны в табл. 18.4.

Вдобавок к методам RegisterChannel() и UnregisterChannel() с их ясными названиями, тип ChannelServices определяет свойство RegisteredChannels. Этот член возвращает массив интерфейсов IChannel, каждый из которых представляет дескриптор соответствующего канала из тех, которые зарегистрированы в данном домене приложения.

Таблица 18.4. Подборка членов типа ChannelServices

|  |  |
| --- | --- |
| **Член** | **Описание** |
| RegisteredChannels | Свойство, получающее или устанавливающее список зарегистрированных в настоящий момент каналов, каждый из которых представляется интерфейсом IChannel |
| DispatchMessage() | Метод, выполняющий обработку поступающих удаленных вызовов |
| GetChannel() | Метод, возвращающий зарегистрированный канал с указанным именем |
| GetUrlsForObject() | Метод, возвращающий массив адресов URL, которые могут использоваться для доступа к указанному объекту |
| RegisterChannel() | Метод, регистрирующий канал о соответствующими канальными сервисами |
| UnregisterChannel() | Метод, отменяющий регистрацию данного канала и удаляющий этот канал из списка зарегистрированных |

Определение интерфейса IChannel оказывается исключительно простым.

publiс interface IChannel {

 string ChannelName { get; }

 int ChannelPriority { get; }

 string Parse(string url, ref String objectURI);

}

Как видите, каждый канал получает понятное строковое имя, а также свой уровень приоритета. Например, если добавить в метод Main() приложения SimpleRemoteObjectClient следующую) программную логику

**// Список всех зарегистрированных каналов.**

IChannel[] сhannelObjs = ChannelServices.RegisteredChannels;

foreach (IChannel i in channelObjs) {

 Console.WriteLine("Имя канала: {0}", i.ChannelName);

 Console.WriteLine("Приоритет: {0}", i.ChannelPriority);

}

то в окне консоли клиента вы увидите вывод, подобный показанному на рис. 18.4.



Рис. 18.4. Список каналов в окне клиента

**Тип RemotingConfiguration**

Другим ключевым типом удаленного взаимодействия является тип RemotingConfiguration, который, в соответствии со своим названием, используется для настройки различных параметров приложения удаленного взаимодействия. Вы уже видели этот тип в работе на стороне сервера (при вызове метода RegisterWellKnownServiceType()). Другие заслуживающие внимания статические члены этого типа описываются в табл. 18.5, а возможности применения некоторых из этих членов будут продемонстрированы в оставшейся части этой главы.

**Таблица 18.5.**Члены типа RemotingConfiguration

|  |  |
| --- | --- |
| **Член** | **Описание** |
| ApplicationId | Возвращает идентификатор приложения, выполняющегося в настоящий момент |
| ApplicationName | Возвращает или устанавливает имя приложения удаленного взаимодействия |
| ProcessId | Возвращает идентификатор процесса, выполняющегося в настоящий момент |
| Configure() | Читает файл конфигурации и устанавливает параметры конфигурации инфраструктуры удаленного взаимодействия |
| GetRegisteredActivatedClientTypes() | Возвращает массив объектных типов, зарегистрированных на стороне клиента для удаленной активизации |
| GetRegisteredActivatedServiceTypes() | Возвращает массив объектных типов, зарегистрированных на стороне сервиса для активизации по запросу клиента |
| GetRegisteredWellKnownClientTypes() | Возвращает массив объектных типов, зарегистрированных на стороне клиента в качестве WKO-типов |
| GetRegisteredWellKnownServiceTypes() | Возвращает массив объектных типов, зарегистрированных на стороне сервиса в качестве WKO-типов |
| IsWellKnownClientType() | Проверяет, является ли указанный объектный тип зарегистрированным WKO-типом клиента |
| RegisterActivatedClientType() | Регистрирует объект на стороне клиента как тип, позволяющий активизацию на сервере |
| RegisterWellKnownClientType() | Регистрирует объект на стороне клиента как WKO-тип (синглет или объект одиночного вызова) |
| RegisterWellKnownServiceType() | Регистрирует объект на стороне сервиса как WKO-тип (синглет или объект одиночного вызова) |

Напомним, что слой удаленного взаимодействия .NET различает два вида MBR-объектов: WKO (активизируются сервером) и САО (активизируются клиентом). К тому же, WKO-тип может быть активизирован либо как синглет, либо как объект одиночного вызова. Используя функциональные возможности типа RemotingConfiguration, вы можете динамически получить такую информацию в среде выполнения. Например, если добавить в метод Main() приложения SimpleRemoteObjectServer следующие строки программного кода:

static void Main(string[] args) {

 …

**// Установка понятного имени для данного приложения сервера.**

 RemotingConfiguration.ApplicationName = "Первое серверное приложение";

 Console.WriteLine("Имя приложения: {0}", RemotingConfiguration.ApplicationName);

**// Получение массива типов WellKnownServiceTypeEntry,**

**// представляющих зарегистрированные WKO-объекты.**

 WellKnownServiceTypeEntry[] WKOs = RemotingConfiguration.GetRegisteredWellKnownServiceTypes();

**// Вывод информации.**

 foreach(WellKnownServiceTypeEntry wko in WKOs) {

  Console.WriteLine("Имя блока, содержащего WKO: {0}", wko.AssemblyName);

  Console.WriteLine("URL данного WKO: {0}", wko.ObjectUri);

  Console.WriteLine("Тип WKO: {0}", wko.ObjectType);

  Console.WriteLine("Режим активизации WKO: {0}", wko.Mоde);

 }

}

то вы должны увидеть список всех WKO-типов, зарегистрированных доменом приложения сервера. Выполнив цикл по всем элементам массива типов WellKnownServiceTypeEntry, можно выяснить характеристики каждого из WKO-объектов. Поскольку ваше серверное приложение регистрирует только один тип (SimpleRemotingAsm.RemoteMessageObject), вы получите вывод, показанный на рис. 18.5.



Рис. 18.5. Статистика сервера

Другим важным методом типа RemotingConfiguration является метод Configure(). Вскоре вы сможете убедиться, что этот статический член позволяет доменам приложений клиента и сервера использовать файлы конфигурации удаленного взаимодействия.

**Снова о режиме активизации WKO-типов**

Напомним, что WKO-типы можно настроить для работы либо в режиме синглета, либо в режиме объекта одиночного вызова. В настоящее время ваше серверное приложение регистрирует WKO-тип с использованием семантики активизации синглета.

**// Синглеты могут обслуживать множество клиентов**.

RemotingConfiguration.RegisterWellKnownServiceType(typeof(SimpleRemotingAsm.RemoteMessageObject), "RemoteMsgObj.soap", **WellKnownObjectMode.Singleton)**;

Снова обратим внимание на то, что WKO-синглеты могут получать запросы от множества клиентов. Поэтому синглеты связаны с удаленными клиентами отношением "один ко множеству". Чтобы проверить это непосредственно, запустите приложение сервера (если оно в настоящий момент еще не выполняется) и три отдельных приложения клиента. Если посмотреть на вывод сервера, вы обнаружите там только один вызов заданного по умолчанию конструктора RemoteMessageObject.

Чтобы рассмотреть поведение объектов одиночного вызова, измените серверное приложение так, чтобы в нем регистрировался WKO-объект, поддерживающий активизацию одиночного вызова.

**// WKO-типы одиночного вызова связаны с клиентом**

**// отношением "один к одному".**

RemotingConfiguration.RegisterWellKnownServiceType(typeof(SimpleRemotingAsm.RemoteMessageObject), "RemoteMsgObj.soap", **WellKnownObjectMode.SingleCall)*;***

После перекомпиляции и запуска серверного приложения снова запустите три клиента. На этот раз вы увидите, что для каждого запроса клиента будет создан новый RemoteMessageObject. Итак, если вы хотите сделать данные состояния общими для множества удаленных клиентов, то активизация синглета оказывается единственным подходящим вариантом, поскольку тогда все клиенты "общаются" с единственным экземпляром удаленного объекта.

**Исходный код.**Проекты SimpleRemotingAsm, SimpleRemoteObjectServer и SimpleRemoteObjectClient размещены в подкаталоге, соответствующем главе 18.

**Установка сервера на удаленной машине**

К этому моменту вы реализовали возможность пересечения границ приложения и процесса на одной машине. Если у вас есть возможность связи с другой машиной, вы можете расширить свой пример так, чтобы клиент мог взаимодействовать с типом RemoteMessageObject через границы машин. Для этого необходимо сделать следующее.

1. На машине сервера создайте и откройте для доступа папку, в которой будут содержаться компоновочные блоки серверной стороны,

2*.*Скопируйте компоновочные блоки SimpleRemoteObjeсtServer.exe и SimpleRemotingAsm.dll в эту папку.

3. Откройте проект SimpleRemoteObjectClient и измените URL активизации в соответствии с именем удаленной машины, например:

// Получение агента для удаленного объекта.

object remoteObj = Activator.GetObject(typeof(SimpleRemotingAsm.RemoteMessageObject), "httр://**ИмяУдаленнойМашины**:32469/RemoteMsgObj.soap");

4. Запустите приложение SimpleRemoteObjectServer.exe на машине сервера.

5. Запустите приложение SimpleRemoteObjectClient.exe на машине клиента.

6. Откиньтесь на спинку кресла, расслабьтесь и улыбнитесь.

**Замечание**. Вместо понятного имени Машины URL активизации может указывать ее IP-адрес.

**Использование ТСР-каналов**

В настоящий момент ваш удаленный объект доступен через сетевой протокол HTTP. Как уже упоминалось выше, этот протокол вполне совместим с брандмауэром, но генерируемые при этом пакеты SOAP немного "раздуты" (по причине представления данных в формате XML). Чтобы уменьшить сетевой трафик, можно изменить компоновочные блоки клиента и сервера так, чтобы в них использовался TCP-канал и, следовательно, тип BinaryFormatter. Вот подходящая модификация компоновочного блока сервера.

**Замечание**. Для файлов с определениями объектов, доступных по TCP-каналам о заданным URI, чаще всего (но не обязательно) используется расширение \*.rem (от remote – удаленный).

**// Корректировки для сервера.**

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

…

static void Main(string[] args) {

 …

**// Создание нового TcpChannel**

 TcpChannel с = new TcpChannel(32469);

 ChannelServises.RegisterChannel(c);

**// Регистрация WKO-объекта в режиме синглета.**

 RemotingConfiguration.RegisterWellKnownServiceType(typeof(SimpleRemotingAsm.RemoteMessageObject), **"RemoteMsgObj**.**rem"**, WellKnownObjectMode.SingleCall);

 Console.ReadLine();

}

Здесь в слое удаленного взаимодействия .NET регистрируется тип System. Runtime.Remoting.Channels.Tcp.TcpChannel. Кроме того, изменен URI-объект (теперь для него задано более общее имя RemoteMsgObj.rem вместо \*.soap, что явно указывало на использование SOAP). Модификация приложения клиента так же проста.

**// Корректировки для клиента.**

**using System.Runtime.Remoting.Channels**.**Тcр**;

…

static void Main(string[] args) {

 …

**// Создание нового TcpChannel**

 TcpChannel с = new TcpChannel();

 ChannelServices.RegisterChannel(c);

**// Получение агента для удаленного объекта.**

object remoteObj = Activator.GetObject(typeof(SimpleRemotingAsm.RemoteMessageObject), **"tcp://localhost:32469/RemoteMsgObj.rem");**

**// Использование объекта.**

 RemoteMessageObject simple = (RemoteMessageObject)remoteObj;

 simple.DisplayMessage("Привет от клиента!");

 Console.WriteLine("Сервер говорит: {0}", simple.ReturnMessage());

 Console.ReadLine();

}

Единственным заслуживающим внимания моментом здесь является то, что URL активизации клиента теперь должен содержать признак канала tcp://, а не http://. Во всем остальном программная логика здесь оказывается идентичной программной логике HttpChannel,

**Исходный код.**Проекты TCPSimpleRemoteObjectServer и TCPSimpleRemoteObjectClient размещены в подкаталоге, соответствующем главе 18 (оба эти проекта используют созданный выше компоновочный блок SimpleRemotingAsm.dll).

**Несколько слов о IpcChannel**

Перед тем как перейти к обсуждению файлов конфигурации удаленного взаимодействия, напомним, что .NET 2.0 предлагает тип IpcChannel, обеспечивающий самый быстрый из возможных способов взаимодействия приложений *на одной*машине. Задачей данной главы является изучение возможностей построения распределенных приложений, выполняемых не на одном, а на множестве компьютеров. Поэтому по поводу использования IpcChannel обратитесь к документации .NET Framework 2.0 SDK (как и следует ожидать, соответствующий программный код будет почти идентичен программному коду, необходимому для работы с HttpChannel и TcpChannel).

**Файлы конфигурации удаленного взаимодействия**

Итак, вы успешно построили распределённое приложение, используя слой удаленного взаимодействия .NET. В связи c данными примерами следует обратить внимание на то что полученные приложения клиента и сервера содержат большой объем "жестко" кодируемой программной логики. Например, сервер указывает фиксированный идентификатор порта, фиксированный режим активизации и фиксированный тип канала. Клиент, с другой стороны, "жестко" кодирует имя удаленного объекта, с которым пытается взаимодействовать.

Согласитесь, слишком наивно предполагать, что исходные параметры проекта смогут оставаться неизменными при любой инсталляции приложения. В идеале такие параметры, как номер порта, режим активизации и другие подобные элементы, должны позволять динамическое изменение без перекомпиляции и переустановки приложений клиента и сервера. В соответствии со схемой удаленного взаимодействия .NET все упомянутые здесь проблемы могут быть решены с помощью файла конфигурации удаленного взаимодействия;.

Вспомните из главы 11 о том, что файл \*.config можно использовать для "подсказок" среде CLR в отношений места нахождения внешних компоновочных блоков, необходимых для работы приложения. Эти же файлы \*.config могут использоваться и для информирования CLR о целом ряде параметров удаленного взаимодействия, как на стороне клиента, так и на стороне сервера.

При создании файла \*.config для указания различных параметров удаленного взаимодействия используют элемент ‹system.runtime.remoting›. Если у вашего приложения уже есть файл \*.config, в котором указаны параметры размещения компоновочного блока, вы можете добавить элементы удаленного взаимодействия в этот же файл. Единый файл \*.config, содержащий и настройки удаленного взаимодействия, и информацию привязки, должен выглядеть примерно так.

‹configuration›

 ‹sуstem.runtime.remoting›

**‹!-- параметры удаленного взаимодействия клиента и сервера --›**

 ‹/system.runtime.remoting*›*

*‹*runtime›

**‹!-- информация привязки компоновочного блока --›**

 ‹/runtime*›*

*‹*/configuration›

Если вам нечего указать в отношении привязки компоновочного блока, вы можете опустить соответствующий элемент ‹runtime› и использовать в файле \*.config шаблон следующего вида.

‹configuration›

*‹*system.runtime.remoting›

**‹!-- параметры удаленного взаимодействия клиента и сервера --›**

 ‹/system.runtime.remoting›

‹/configuration›

**Создание файлов \*.config сервера**

Файлы конфигурации на стороне сервера позволяют объявить объекты, которые будут доступны для удаленных вызовов, а также задать параметры канала и порта. Рассмотрим следующий вариант программной логики сервера.

**// "Жестко" заданная программная логика сервера HTTP.**

HttpChannel с = new HttpChannel(32469);

ChannelServices.RegisterChannel(с);

RemotingConfiguration.RegisterWellKnownServiceType(typeof(SimpleRemotingAsm.RemoteMessageObject), "RemoteMsgObj.soap", WellKnownObjectMode.Singleton);

Используя элементы ‹service›, ‹wellknown› и ‹channels›, эту программную логику можно заменить следующим файлом \*.config.

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

   ‹service›

**‹wellknown mode="Singleton" type="SimpleRemotingAsm.RemoteMessageObject, SimpleRemotingAsm" objectUri="RemoteMsgObj.soap"/›**

   ‹/service›

   ‹channels›

**‹channelref="http"/›**

   ‹/channels›

  ‹/application›

 ‹/system.runtime.remoting›

‹/configuration›

Обратите внимание на то, что значительная часть информации удаленного сервера указывается в контексте элемента ‹service› (не *сервер!*). Его дочерний элемент ‹wellknown› использует три атрибута (mode, type и objectUri) для регистрации WKO-объекта в слое удаленного взаимодействия .NET. Элемент ‹channels› может содержать любое число элементов ‹channel›, которые позволяют определить вид канала (в данном случае это HTTP), открываемого на сервере. Для ТСР-каналов вместо http нужно просто использовать лексему tcp.

Поскольку в этом случае вся необходимая информация содержится в файле SimpleRemoteObjectServer.exe.config, метод Main() серверной стороны значительно упрощается. В нем остается выполнить только вызов RemotingConfiguration.Configure() и указать имя соответствующего файла конфигурации.

static void Main(string[] args) {

**// Регистрация WKO-объекта с помощью файла \*.config.**

 RemotingConfiguration.Configure("SimpleRemoteObjectServer.exe.config");

 Console.WriteLine("Старт сервера! Для остановки нажмите ‹Enter›");

 Console.ReadLine();

}

**Создание файлов \*.config клиента**

Клиенты тоже могут использовать файлы *\**.config удаленного взаимодействия. В отличие от файлов конфигурации сервера, в файлах конфигурации клиента для идентификации имени WKO-объекта используется элемент ‹client›. Вдобавок к возможности динамического изменения параметров удаленного взаимодействия без перекомпиляции базового программного кода, файлы \*.config клиента позволяют создать тип агента непосредственно с помощью ключевого слова C# new, не используя метод Activator.GetObject(). Предположим, например, что у нас есть файл \*.config клиента со следующим содержимым.

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

**‹client displayName = "SimpleRemoteObjectClient"›**

**‹wellknown type=" SimpleRemotingAsm.RemoteMessageObject, SimpleRemotingAsm" url="http://localhost:32469/RemoteMsgObj.soap"/›**

**‹/client›**

   ‹channels›

**‹channel ref="http"/›**

   ‹/channels›

  ‹/application›

 ‹/system.runtime.remoting›

‹/configuration›

Тогда можно изменить метод Main() клиента так.

statiс void Main(string[] args) {

 RemotingConfiguration.Configure("SimpleRemoteObjectClient.exe.config");

**// При использовании файла \*.config клиент может создать тип**

**// непосредственно с помощью ключевого слова 'new'**.

 RemoteMessageObject simple = new RemoteMessageObject();

 simple.DisplayMessage("Привет от клиента!");

 Console.WriteLine("Сервер говорит: {0}", simple.ReturnMessage());

 Console.WriteLine("Старт клиента! Для остановки нажмите ‹Enter›");

 Console.ReadLine();

}

При выполнении этого варианта приложения вывод оказывается аналогичным исходному. Если клиент пожелает использовать TCP-канал, то для свойств url элемента ‹wellknown› и ref элемента ‹сhannel› следует вместо http указывать tcp.

**Исходный код**. Проекты SimpleRemoteObjectServerWithConfig и SimpleRemoteObjectClientWithConfig размещены в подкаталоге, соответствующем главе 18 (оба эти проекта используют созданный выше компоновочный блок SimpleRemotingAsm.dll).

**Работа с MBV-объектами**

Наши первые приложения удаленного взаимодействия позволяли доступ клиентов к одному WKO-типу. Напомним, что WKO-типы (по определению) являются MBR-типами, поэтому доступ клиента к ним осуществляется через агента-посредника. В противоположность этому, MBV-типы являются локальными копиями серверного объекта, обычно возвращаемыми открытыми членами некоторого MBR-типа. Вы уже знаете, как настроить MBV-тип (следует обозначить соответствующий класс атрибутом [Serializable]), но MBV-тип в действии вы еще не видели (если не считать обмена строковыми данными между двумя сторонами). Для иллюстрации взаимодействия MBR- и MBV-типов мы рассмотрим новый пример, в котором используются следующие три компоновочных блока.

• Общий компоновочный блок CarGeneralAsm.dll

• Компоновочный блок клиента CarProviderClient.exe

• Компоновочный блок сервера CarProviderServer.exe

Как вы можете догадаться, программный код приложений клиента и сервера более или менее подобен программному коду соответствующих приложений предыдущего примера, особенно в том, как эти приложения используют файлы \*.config, Тем не менее, давайте разберем соответствующий процесс построения каждого из указанных компоновочных блоков по очереди.

**Создание общего компоновочного блока**

В ходе нашего обсуждения процесса сериализации объектов в главе 17 мы создали тип JamesBondCar (в дополнение к связанным классам Radio и Car). Библиотека программного кода CarGeneralAsm.dll будет использовать эти типы, поэтому сначала выберите Projects→Add Existing Item из меню и добавьте в свой новый проект библиотеки классов соответствующие файлы \*.cs (автоматически созданный файл Class1.cs можете удалить), Поскольку каждый из добавленных типов уже обозначен атрибутом [Serializable], они готовы для маршалинга по значению в отношении удаленного клиента.

Теперь нам нужен MBR-тип, который обеспечит доступ к типу JamesBondCar. Чтобы сделать ситуацию немного более интересной, ваш MBR-объект (CarProvider) будет поддерживать обобщенный список List‹› типов JamesBondCar. Тип CarProvider определит два члена, которые позволят вызывающей стороне получить заданный тип JamesBondCar, а также полный перечень List‹› соответствующих типов. Вот весь программный код для нового типа класса.

namespace CarGeneralAsm {

**// Этот тип является MBR-объектом, обеспечивающим доступ**

**// к соответствующим MBV-типам.**

 public class CarProvider: MarshalByRefObject {

  private List‹JamesBondCar› theJBCars = new List‹JamesBondCar›();

**// Добавление в список нескольких машин.**

  public CarProvider() {

   Console.WriteLine("Создание поставщика машин");

   theJBCars.Add(new JamesBondCar("QMobile", 140, true, true"));

   theJBCars.Add(new JamesBondCar("Flyer", 140, true, false));

   theJBCars.Add(new JamesBondCar("Swimmer", 140, false, true));

   theJBCars.Add(new JamesBondCar("BasicJBC", 140, false, false));

  }

**// Получение всех JamesBondCar.**

  public List‹JamesBondCar› GetAllAutos() { return theJBCars; }

**// Получение одного JamesBondCar,**

  public JamesBondCar GetJBCByIndex(int i) { return (JamesBondCar)theJBCars[i]; }

 }

}

Обратите внимание на то, что метод GetAllAutos() возвращает внутренний тип List‹›. Очевидный вопрос: как данный член пространства имен System. Collections.Generic представляется вызывающей стороне? Если посмотреть описание этого типа в документации .NET Framework 2.0 SDK, вы обнаружите, что list‹› сопровождается атрибутом [Serializable].

[SerializableAttribute()]

public class List‹T›: IList, ICollection, IEnumerable

Таким образом, для всего содержимого типа List‹› будет использован маршалинг по значению (если содержащиеся в нем типы также допускают сериализацию). Это очень удобная особенность удаленного взаимодействия .NET и членов библиотек базовых классов. Вдобавок к пользовательским MBV- и MBR-типам, которые вы можете создать сами, любой тип из библиотек базовых классов, сопровождающийся атрибутом [Serializable], также способен выступать в качестве MBV-типа в архитектуре удаленного взаимодействия .NET. Аналогично, любой тип, получающийся (непосредственно или косвенно) из MarshalByRefObject, может функционировать, как MBR-тип.

**Замечание**. Следует знать о том, что SoapFormatter не поддерживает сериализацию обобщенных типов. При создании методов, получающих или возвращающих обобщенные типы (напри-мер, List‹›), вы должны использовать BinaryFormatter и объект TcpChannel.

**Создание компоновочного блока сервера**

Компоновочный блок сервера (CarProviderServer.exe) в рамках метода Main() содержит следующую программную логику.

using System;

using System.Runtime.Remoting;

using System.Runtime.Remoting.Channels;

using System.Runtime.Remoting.Channels.Http;

**using CarGeneralAsm;**

namespace CarProviderServer {

 class CarServer {

  static void Main(string[] args) {

   RemotingConfiguration.Configure("CarProviderServer.exe.config");

   Console.WriteLine("Старт сервера! Для остановки нажмите ‹Enter›");

   Console.ReadLine();

  }

 }

}

Соответствующий файл \*.config почти идентичен файлу \*.config сервера, созданному в предыдущем примере. Единственным заслуживающим внимания моментом здесь является определение значения URI объекта для типа CarProvider.

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

   ‹service›

    ‹wellknown mode="Singleton" type="CarGeneralAsm.CarProvider, CarGeneralAsm" objectUri="carprovider.rem" /›

   ‹/service›

   ‹channels›

    ‹channel ref="tcp" port="32469" /›

   ‹/channels›

  ‹/application›

 ‹/system.runtime.remoting›

‹/configuration›

**Создание компоновочного блока клиента**

Наконец, рассмотрим приложение клиента, которое будет использовать MBR-тип CarProvider для получения отдельных типов JamesBondCars и типа List‹›. После получения типа от CarProvider вы посылаете его вспомогательной функции UseCar() для обработки.

using System;

using System.Runtime.Remoting;

using System.Runtime.Remoting.Channels;

using System.Runtime.Remoting.Channels.Http;

**using CarGeneralAsm;**

using System.Collections.Generic;

namespace CarProviderClient {

 class CarClient {

  private static void UseCar(JamesBondCar c) {

   Console.WriteLine("-› Имя: {0}", с.PetName);

   Console.WriteLine("-› Макc. скорость: {0} ", с.MaxSpeed);

   Console.WriteLine("-› Способность плавать: {0}", с.isSeaWorthy);

   Console.WriteLine("-› Способность летать: {0}, c.isFlightWorthy);

   Console.WriteLine();

  }

  static void Main(string[] args) {

   RemotingConfiguration.Configure("CarProviderClient.exe.config");

**// Создание поставщика машин.**

   CarProvider cр = new CarProvider();

**// Получение первого объекта JBC.**

   JamesBondCar qCar = cp.GetJBCByIndex(0);

**// Получение всех объектов JBC.**

   List‹JamesBondCar› allJBCs = cp.GetAllAutos();

**// Использование первой машины.**

   UseCar(gCar);

**// Использование всех машин в List‹›.**

   foreach(JamesBondCar j in allJBCs) UseCar(j);

   Console.WriteLine(''Старт клиента! Для остановки нажмите ‹Enter›");

   Console.ReadLine();

  }

 }

}

Содержимое файла \*.config на стороне клиента также соответствует ожиданиям. Здесь нужно просто изменить URL активизации.

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

   ‹client displayName = "CarClient"›

    ‹wellknown type= "CarGeneralAsm.CarProvider, CarGeneralAsm" url="tcp://localhost:32469/carpovider.rem"/›

   ‹/client›

*‹*channels*›*

    ‹channel ref="http" /›

   ‹/channels›

  ‹/application›

 ‹/system.runtime.remoting›

*‹*/configuration*›*

Теперь запустите свои приложения сервера и клиента (конечно же, в указанном порядке) и рассмотрите соответствующий вывод. В окне консоли клиента будут представлены объекты JamesBondCar и соответствующая информация для каждого типа. Напомним, что вы взаимодействуете с List‹› и типами JamesBondCar, поэтому вы работаете с их членами в рамках домена приложения клиента, так как оба указанных типа обозначены атрибутом [Serializable].

Чтобы доказать это, измените вспомогательную функцию UseCar() так, чтобы она вызывала метод TurnOnRadio() для входного объекта JamesBondCar. Теперь запустите приложения сервера и клиента еще раз. Обратите внимание на то, что на машине клиента теперь появляются соответствующие сообщения. Если бы типы Car, Radio и JamesBondCar были сконфигурированы, как MBR-типы, сообщения бы появлялись на сервере. Для проверки получите каждый из указанных типов из MarshalByRefObject и перекомпилируйте все три компоновочных блока (для гарантии того, что Visual Studio 2005 скопирует самый последний CarGeneralAsm.dll в каталоги приложений клиента и сервера). Теперь при выполнении приложения окно с сообщением появится на удаленной машине.

**Исходный код.**Проекты CarGeneralAsm, CarProviderServer и CarProviderClient размещены в подкаталоге, соответствующем главе 18.

**Объекты, активизируемые клиентом**

Все созданные до сих пор примеры удаленного взаимодействия использовали WKO-типы. Напомним, что WKO-типы имеют следующие особенности.

• WKO-тип можно сконфигурировать как синглет или как объект одиночного вызова.

• WKO-тип можно активизировать только с помощью конструктора типа, заданного по умолчанию.

• Экземпляр WKO-типа создается на сервере при первом запросе члена этого типа клиентом,

Экземпляры САО-типов, с другой стороны, можно создавать с помощью любого конструктора типа, и они создаются тогда, когда клиент использует ключевое слово C# new или тип Activator. Цикл существования САО-типов контролируется механизмом лизингового управления .NET. Следует знать, что при конфигурации САО-типа слой удаленного взаимодействия .NET генерирует специальный САО-объект удаленного взаимодействия для обслуживания каждого клиента. Важной особенностью САО-объектов является то, что они продолжают существовать после завершения вызова отдельного метода (и, таким образом, являются объектами, кумулятивно изменяющими параметры своего состояния в процессе выполнения вызовов клиентов).

Чтобы проиллюстрировать соответствующую конструкцию и использование САО-типов, модифицируем наш уже имеющийся "автомобильный" компоновочный блок. В нашем MBR-классе CarProvider определим дополнительный конструктор, позволяющий клиенту передать массив типов JamesBondCar, предназначенных для размещения в обобщенном списке List‹›.

public class CarProvider: MarshalByRefObject {

 private List‹JamesBondCar› theJBCars = new List‹JamesBondCar›();

 public CarProvider**(JamesBondCar[] theCars)**{

  Console.WriteLine("Создание поставщика Car");

  Console.WriteLine("с помощью пользовательского конструктора');

  theJBCars.AddRange(theCars);

 }

 …

}

Чтобы позволить вызывающей стороне активизировать CarProvider с помощью нового конструктора, нужно построить приложение сервера, которое зарегистрирует CarProvider, как САО-тип, а не как WKO-тип. Это можно сделать программно с помощью метода, аналогичного RemotingConfiguration.RegisterActivatedServiceType(), или с помощью файла \*.config на стороне сервера. Чтобы "жестко" задать имя CAO-объекта в программном коде сервера, передайте информацию типа или типов (после создания и регистрации канала), как предлагается ниже.

**// "Жёсткое" указание того, что CarProvider является САО-типом.**

RemotingConfiguration.RegisterActivatedServiceType(typeof(CAOCarGeneralAsm.CarProvider));

Если вы предпочтете использовать файл \*.config, вместо элемента ‹wellknown› используйте элемент ‹activated›, как показано ниже.

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

   ‹service›

**‹activated type = "CAOCarGeneralAsm.CarProvider, CAOCarGeneralAsm"**/›

   ‹/service›

   ‹channels›

    ‹channel ref="tcp" port="32496" /›

*‹*/channels›

*‹/*application›

 ‹/system.runtime.remoting›

‹/configuration›

Наконец, нужно обновить приложение клиента, и не только с целью учета соответствующего файла \*.config (или программных изменений в базовом коде) для запроса доступа к удаленному САО-объекту, но и с тем, чтобы вызвать созданный пользовательский конструктор типа CarProvider. Вот как должен выглядеть модифицированный метод Main() на стороне клиента.

static void Main(string[] args) {

**// Чтение обновленного файла \*.config.**

 RemotingConfiguration.Configure("CAOCarProviderClient.exe.config");

**// Создание массива типов для передачи поставщику.**

 JamesBondCar[] cars = {

  new JamesBondCar ("Viper", 100, true, false),

  new JamesBondCar("Shaken", 100, false, true),

  new JamesВоndCar("Stirred", 100, true, true)

 };

**// Теперь вызов пользовательского конструктора.**

 CarProvider ср = new CarProvider(cars);

 …

}

Обновленный файл \*.сonfig клиента также должен использовать элемент ‹activated›, а не элемент ‹wellknown›. Кроме того, свойство url элемента ‹client› теперь должно указывать адрес зарегистрированного САО-объекта. Напомним, что при регистрации типа CarProvider сервером в виде WKO-объекта, клиент указывал соответствующую информацию в рамках элемента ‹wellknown›.

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

   ‹client displayName = "CarClient" url = "tcp://localhost:32469"›

**‹activated type="CAOCarGeneralAsm.CarProvider, CAOCarGeneralAsm"**/›

   ‹/client›

   ‹channels›

    ‹channel ref="tcp"/›

   ‹/channels›

  ‹/application›

 ‹/system.runtime.remoting›

‹/configuration›

Чтобы "жестко" запрограммировать запрос САО-типа клиентом, можете использовать метод RegistrationServices.RegisterActivatedClientType(), как показано ниже.

static void Main(string[] args) {

**// Использование "жестких" значений.**

 RemotingConfiguration.RegisterActivatedClientType(typeof(CAOCarGeneralAsm.CarProvider), "tcp://localhost:32469");

}

Запустив на выполнение обновленные компоновочные блоки сервера и клиента, вы с удовлетворением обнаружите, что можете передать свой пользовательский массив типов JamesBondCar удаленному объекту CarProvider через перегруженный конструктор.

**Исходный код.**Проекты CAOCarGeneralAsm, CAOCarProviderServer и CAOCarProviderCIient размещены в подкаталоге, соответствующем главе 18.

**Схема лизингового управления циклом существования САО-типов и WKO-синглетов**

Вы уже видели, что WKO-типы, сконфигурированные для активизации одиночного вызова, существуют только в процессе текущего вызова метода. Поэтому WKO-типы одиночного вызова являются объектами, не меняющими своего состояния в процессе выполнения. Как только текущий вызов завершается, WKO-тип одиночного вызова становится объектом, предназначенным для участия в очередной процедуре сборки мусора.

С другой стороны, САО-типы, а также WKO-типы, сконфигурированные для активизации в виде синглета, являются по своей природе объектами, кумулятивно изменяющими параметры своего состояния в процессе выполнения вызовов клиентов. Учитывая эти две доступные опции установки конфигурации, возникает следующий вопрос: как процесс сервера "узнает" о том, что пора уничтожить такой MBR-объект? Если сборщик мусора на сервере уничтожит MBR-объекты, находящиеся в использовании удаленным клиентом, это создаст проблемы, А если серверу придется ожидать освобождения MBR-типов слишком долго, это отрицательно повлияет на работу системы, особенно если соответствующие MBR-объекты удерживают важные ресурсы (связь с базой данных, неуправляемые типы или какие-то другие ресурсы).

Цикл существования MBR-объекта, являющегося CAO-типом или WKD-синглетом, контролируется по схеме лизингового управления, которая тесно связана с процессом сборки мусора .NET. Если "время аренды" MBR-объекта, являющегося CAO типом или WKO-синглетом истекает, объект становится кандидатом на участие в очередном цикле сборки мусора. Как и в случае любого другого .NET-типа, если удаленный объект переопределяет System.Object.Finalize() (с помощью синтаксиса деструктора C#), то среда выполнения .NET автоматически запустит соответствующую логику финализации.

**Схема лизингового управления, используемая по умолчанию**

Для MBR-объектов, являющихся САО-типами или WKO-синглетами, применяется так называемый *лизинг по умолчанию,*время которого равно пяти минутам. Если среда выполнения обнаружит, что MBR-объект, являющийся САО-типом или WKO-синглетом, остается неактивным в течение пяти минут, делается вывод о том, что клиент больше не использует данный удаленный объект, и поэтому этот объект может использоваться в процессе сборки мусора. При этом совсем не обязательно, чтобы после истечения времени лизинга объект помечался для сборки мусора *немедленно.*На самом деле есть много возможностей влиять на поведение, задаваемое лизингом по умолчанию.

Например, при каждом вызове клиентом члена удаленного MBR-обьекта, являющегося САО-типом или WKO-синглетом, время лизинга снова устанавливается равным пяти минутам. Но кроме автоматического обновления интервала времени лизинга при вызове клиента, среда выполнения .NET обеспечивает три дополнительные альтернативы.

• Установки лизинга по умолчанию для удаленных объектов могут переопределяться файлами \*.config.

• Могут использоваться спонсоры лизинговой схемы сервера, действующие от имени удаленного объекта, время лизинга которого уже истекло.

• Могут использоваться спонсоры лизинговой схемы клиента, действующие от имени удаленного объекта, время лизинга которого уже истекло.

Мы рассмотрим каждую из указанных возможностей в следующих разделах, а пока что давайте рассмотрим установки лизинга, принятые для удаленного типа по умолчанию. Вспомните, что базовый класс MarshalByRefObject определяет член с именем GetLifetimeService(). Этот метод возвращает ссылку на внутренний объект, поддерживающий интерфейс System.Runtime.Remoting.Lifetime.ILease. Интерфейс ILease можно использовать для управления параметрами лизинга данного САО-типа или WKO-синглета. Вот формальное определение этого интерфейса.

public interface **ILease**{

 TimeSpan CurrentLeaseTime { get; }

 LeaseState CurrentState { get; }

 TimeSpan InitialLeaseTime { get; set; }

 TimeSpan RenewOnCallTime { get; set; }

 TimeSpan SponsorshipTimeout { get; set; }

 void Register(System.Runtime.Remoting.Lifetime.ISponsor obj);

 void Register(System.Runtime.Remoting.Lifetime.ISponsor obj, TimeSpan renewalTime);

 TimeSpan Renew(TimeSpan renewalTime);

 void Unregister(System.Runtime.Remoting.Lifetime.ISponsor obj);

}

Интерфейс ILease не только позволяет получить информацию о текущих параметрах лизинга (с помощью CurrentLeaseTime, CurrentState и InitialLeaseTime), но и обеспечивает возможность построения "спонсоров" лизинга (более подробно об этом будет говориться позже). Роль каждого из членов ILease описана в табл. 18.6.

**Таблица 18.6.**Члены интерфейса ILease

|  |  |
| --- | --- |
| **Член** | **Описание** |
| CurrentLeaseTime | Читает информацию о времени, оставшемся до отключения данного объекта при отсутствии новых вызовов методов объекта |
| CurrentState | Читает информацию о текущем состоянии лизинга, представленную значением перечня LeaseState |
| InitialLeaseTime | Читает или устанавливает исходное время лизинга. Исходное время лизинга – это время от начала активизации объекта до истечения лизинга при отсутствии новых вызовов методов объекта |
| RenewOnCallTime | Читает или устанавливает значение времени, на которое вызов удаленного объекта увеличивает значение CurrentLeaseTime |
| SponsorshipTimeout | Читает или устанавливает значение времени ожидания спонсора для возвращения времени возобновления лизинга |
| Register() | Перегруженный метод, регистрирующий спонсора данного лизинга |
| Renew() | Возобновляет лизинг с указанным временем |
| Unregister() | Удаляет указанный спонсор из списка спонсоров |

Для иллюстрации особенностей лизинга по умолчанию для удаленных СAО-типов и WKO-синглетов определим в нашем текущем проекте CAOCarGeneralAsm новый внутренний класс LeaseInfo. Статический член LeaseStats() этого класса выводит информацию о текущем лизинге для типа CarProvider в окно консоли сервера (не забудьте указать директиву using для пространства имен System.Runtime.Remoting.Lifetime, чтобы сообщить компилятору о месте нахождения определении типа ILease).

internal class LeaseInfo {

 public static void LeaseStats(ILease itfLease) {

  Console.WriteLine(\*\*\*\*\* Информация о лизинге \*\*\*\*\*");

  Console.WriteLine("Состояние лизинга: {0}", itfLease.CurrentState);

  Console.WriteLine("Начальное время лизинга: {0}:{1}", itfLease.InitialLeaseTime.Minutes, itfLease.InitialLeaseTime.Seconds);

  Console.WriteLine("Текущее время лизинга: {0}:{1}", itfLease.CurrentLeaseTime.Minutes, itfLease.CurrentLeaseTime.Seconds);

  Console.WriteLine("Обновление времени при вызове: {0}:{1}", itfLease.RenewOnCallTime.Minutes, itfLease.RenewOnCallTime.Seconds);

  Console.WriteLine();

 }

}

Теперь предположим, что LeaseInfo.LeaseStats() вызывается в рамках методов GetJBCByIndex() и GetAllAutos() типа CarProvider. После перекомпиляции компоновочных блоков сервера и клиента (снова для гарантии того, что система Visual Studio 2005 скопирует самую последнюю и наиболее полную версию CarGeneralAsm.dll в каталоги приложений клиента и сервера), выполните приложение еще раз. Окно консоли вашего сервера должно теперь быть похожим на то, которое показано на рис. 18.6.

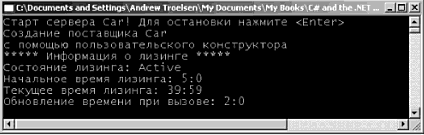


Рис. 18.6. Информация лизинга по умолчанию для CarProvider

**Изменение параметров схемы лизингового управления**

Очевидно, параметры лизинга по умолчанию не могут годиться во всех случаях и для всех удаленных САО-объектов и WKO-синглетов. Если вы хотите изменить типовые установки, у вас на выбор есть два варианта.

• Установки лизинга, принятые по умолчанию, можно изменить с помощью файла \*.config сервера.

• Установки лизинга, принятые по умолчанию, можно изменить программными средствами путем переопределения членов базового класса MarshalByRefObject.

Каждый из этих подходов действительно позволяет изменить установки лизинга, принятые по умолчанию, но между этими подходами есть принципиальная разница. При использовании файла \*.config сервера установки лизинга применяются ко *всем*объектам, размещаемым в рамках процесса сервера. При переопределении отдельных членов типа MarshalByRefObject появляется возможность изменять установки лизинга для каждого объекта в отдельности.

Чтобы продемонстрировать изменение параметров лизинга по умолчанию с помощью файла \*.config, добавим к XML-данным сервера дополнительный элемент ‹lifetime›.

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

**‹lifetime leaseTime = "15M" renewOnCallTime**= **"5M"/›**

   ‹service›

    ‹activated type="CarGeneralAsm.CarProvider, CarGeneralAsm" /›

   ‹/service›

   ‹channels›

    ‹channel ref="tcp" port="32469" /›

   ‹/channels›

  ‹/application

 ‹/system.runtime.remoting›

‹/configuration›

Обратите внимание на то, что в значениях свойств leaseTime и renewOnCallTime используется суффикс M, который, как вы можете догадаться сами, при установке времени для лизинга обозначает использование минут в качестве единицы измерения. При необходимости числовые значения элемента ‹lifetime› могут также содержать суффиксы MS (миллисекунды), S (секунды), Н (часы) и даже D (дни).

Повторим, что при изменении файла \*.config сервера вы изменяете параметры лизинга для каждого САО-объекта и WKO-синглета в рамках сервера. Как альтернативу, можно использовать программное переопределение метода InitializeLifetime() конкретного удаленного типа.

public class CarProvider: MarshalByRefObject {

 public override object InitializeLifetimeService() {

**// Получение текущей информации лизинга.**

  ILease itfLeaseInfo = (ILease)base.InitializeLifetimeService();

**// Изменение установок.**

  itfLeaseInfo.InitialLeaseTime = TimeSpan.FromMinutes(50);

  itfLeaseInfo.RenewOnCallTime = TimeSpan.FromMinutes(10);

  return itfLeaseInfo;

 }

 …

}

Здесь CarProvider устанавливает значение 50 минут для InitialLeaseTime и значение 10 – для RenewOnCallTime. Снова подчеркнем, что преимуществом переопределения метода InitializeLifetimeServices() является возможность, настройки каждого удаленного типа в отдельности.

Наконец, чтобы вообще отключить ограничения для времени лизинга данного СAО-типа или WKO-синглета, переопределите InitializeLifetimeServices() так, чтобы возвращалось значение null. В результате вы, по сути, укажете МВR-тип, который будет существовать всё время, пока будет работать хост-приложение сервера.

**Настройка параметров лизинга на стороне сервера**

Вы только видели, что переопределение метода InitializeLifetimeServices() MBR-типом позволяет изменить текущие параметры лизинга во время активизации типа. Но что делать, если удаленному типу нужно изменить параметры лизинга *после*активизации? Предположим, например, что тип CarProvider предлагает новый метод, выполняющий операцию, требующую много времени (например; соединение с удаленной базой данных с последующим чтением большого набора записей). Перед началом выполнения такого заданий вы можете программно изменить время лизинга так, чтобы в случае, когда остаток времени становится менее одной минуты, время лизинга снова увеличивалось до десяти минут. Для этого можно использовать наследуемые методы MarshalByRefObject.GetLifetimeService() и ILease.Renew() так, как предлагается ниже.

**// Корректировка параметров лизинга на стороне сервера.**

**// Предполагается, что это новый метод типа CarProvider.**

public void DoLengthyOperation() {

 ILease itfLeaseInfo = (ILease)this.GetLifetimeService();

 if (itfLeaseInfo.CurrentLeaseTime.TotalMinutes ‹ 1.0) itfLeaseInfo.Renew(TimeSpan.FromMiutes(10));

**// Выполнение длительной операции…**

}

**Настройка параметров лизинга на стороне клиента**

В дополнение к указанным возможностям ILease, домен приложения клиента тоже может регулировать текущие параметры лизинга CAO-типов и WKD-сингле-тов, с которыми осуществляется удаленное взаимодействие. Для этого клиент должен использовать статический метод RemotingServices.GetLifetimeService(). В качестве параметра указанному члену клиент должен передать ссылку на удаленный тип так, как показано ниже.

**// Корректировка параметров лизинга на стороне клиента.**

CarProvider ср = new CarProvider(сars);

ILease itfLeaseInfo = (ILease)RemotingServices.GetLifetimeServiсе(cp);

if (itfLeaseInfo.CurrentLeaseTime.TotalMinutes ‹ 10.0) itfLeaseInfo.Renew(TimeSpan.FromMinutes(1000));

Такой подход может быть полезен тогда, когда домен приложения клиента готов начать выполнение длительной операции в потоке, использующем удаленный тип. Например, если однопоточное приложение должно напечатать документ, содержащий 100 страниц текста, очень велика вероятность того, что удаленный САО-тип или WKO-синглет может выйти за рамки отведенного для процесса времени. Надеюсь, вы уловили общую идею, хотя здесь, конечно, более "элегантным" решением является создание нового потока выполнения.

**Спонсоры лизинга сервера (и клиента)**

Заключительной темой нашего связанного с лизингом обсуждения цикла существования САО-типов и WKO-синглетов будет *спонсорство лизинга.*Как вы только что убедились, для каждого объекта САО-типа и WKO-синглета имеются параметры лизинга, используемые по умолчанию, которые можно изменить несколькими способами, как на стороне сервера, так и на стороне клиента. Но, независимо от конфигурации лизинга типа, в конечном итоге время лизинга MBR-объекта истечет. В этот момент среда выполнения отправит данный объект в "мусорник"… ну, хорошо, почти отправит.

Суть в том, что перед тем, как пометить ненужный тип для отправки сборщику мусора, среда выполнения проверяет, не имеет ли данный MBR-объект зарегистрированных спонсоров лизинга. Простыми словами, спонсор – это тип, реализующий интерфейс ISponsor, который определен так, как показано ниже.

public interface System.Runtime.Remoting.Lifetime.**ISponsor**{

 TimeSpan Renewal(ILease lease);

}

Если среда выполнения обнаружит, что у MBR-объекта имеется спонсор, этот объект *не будет*сразу же отправлен сборщику мусора, а будет вызван метод Renewal() объекта спонсора, чтобы (еще раз) добавить время к текущему времени лизинга. С другой стороны, если окажется, что для данного MBR-типа спонсора нет, цикл существования объекта действительно закончится.

Предположим, что вы создали пользовательский класс, реализующий ISponsor и вызывающий метод Renewal() для возврата конкретной величины времени (через тип TimeSpan). Тогда как ассоциировать указанный тип с данным удаленным объектом? И снова это может быть сделано либо доменом приложения сервера, либо доменом приложения клиента.

Для этого заинтересованная сторона должна получить ссылку ILease (с помощью наследуемого метода GetLifetimeService() на стороне сервера или статического метода RemotingServices.GetLifetimeService() на стороне клиента) и вызвать Register().

**// Регистрация спонсора на стороне сервера.**

CarSponsor mySponsor = new CarSponsor();

ILease itfLeaseInfo = (ILease)this.GetLifetimeService();

itfLeaseInfo.Register(mySponsor);

**// Регистрация спонсора на стороне клиента.**

CarSponsor mySponsor = new CarSponsor();

CarProvider cp = new CarProvider(cars);

ILease itfLeaseInfo = (ILease)Remoting.Services.GetLifetimeService(cp);

itfLeaseInfo.Register.(mySponsor);

В любом случае, если клиент или сервер желают отменить спонсорство, это можно сделать с помощью метода ILease.Unregister(), например:

**// Отключение спонсора для данного объекта.**

itfLeaseInfo.Unregister(mySponsor);

**Замечание**. Объекты клиента, имеющие спонсоры, кроме необходимости реализации ISponsor должны быть производными от MarshalByRefObject, поскольку клиент должен передать спонсор в удаленный домен приложения.

Как видите, управление циклом существования MBR-типов, кумулятивно изменяющих параметры своего состояния в процессе выполнения вызовов клиентов, оказывается немного более сложным, чем простая сборка мусора. На стороне преимуществ мы имеем *широкие*возможности управления относительно того, когда именно следует уничтожить удаленный тип. С другой стороны, существует вероятность того, что удаленный тип будет уничтожен без ведома клиента. Если клиент попытается вызвать члены типа, уже удаленного из памяти, среда выполнения сгенерирует исключение System.Runtime.Remoting.RemotingException, и в этот момент клиент может либо создать новый экземпляр удаленного типа, либо выполнить другие предусмотренные для такого случая действия.

**Исходный код**. Проекты CAOCarGeneralAsmLease, CAOCarProviderServerLease и CAOCarProviderClientLease размещены в подкаталоге, соответствующем главе 18.

**Альтернативные хосты для удаленных объектов**

При изучении материала этой главы вы создали группу консольных серверных хостов, обеспечивающих доступ к некоторому множеству удаленных объектов. Если вы имеете опыт использования классической модели DCOM (Distributed Component Object Model – распределенная модель компонентных объектов), соответствующие шаги могут показаться вам немного странными. В мире DCOM обычно строится один COM-сервер (на стороне сервера), содержащий удаленные объекты, который несет ответственность и за прием запросов, поступающих от удаленного клиента. Это единственное DCOM-приложение \*.exe "спокойно" загружается в фоновом режиме без создания, в общем-то ненужного командного окна.

При построении компоновочного блока сервера .NET велика вероятность того, что удаленной машине не придется отображать никаких сообщений. Скорее всего, вам понадобится сервер, который только откроет подходящие каналы и зарегистрирует удаленные объекты для доступа клиента. Кроме того, при наличии простого консольного хоста вам (или кому-нибудь другому) придется вручную запустить компоновочный блок \*.exe на стороне сервера, поскольку система удаленного взаимодействия .NET не предусматривает автоматический запуск файла \*.exe на стороне сервера при вызове удаленным клиентом.

С учетом этих проблем возникает следующий вопрос: как создать невидимый приемник, который загрузится автоматически? Программисты .NET предлагают здесь на выбор две возможности.

• Построение .NET-приложения сервиса Windows, готового предложить хостинг для удаленных объектов.

• Разрешение осуществлять хостинг для удаленных объектов серверу IIS (Internet Information Server – информационный сервер Интернет).

**Хостинг удаленных объектов с помощью сервиса Windows**

Возможно, идеальным хостом для удаленных объектов является сервис Windows, поскольку сервис Windows позволяет следующее.

• Может загружаться автоматически при запуске системы

• Может запускаться, как "невидимый" процесс в фоновом режиме

• Может выполняться от имени конкретной учетной записи пользователя

Создать пользовательский сервис Windows средствами .NET исключительно просто, особенно в сравнении с возможностями непосредственного использования Win32 API. Для примера мы создадим проект Windows Service с именем CarWinService (рис. 18.7), который будет осуществлять хостинг удаленных типов, содержащихся в CarGeneralAsm.dll.

В результате Visual Studio 2005 сгенерирует парциальный класс (названный по умолчанию Service1), полученный из System.ServiceProcess.ServiceBase, и еще один класс (Program), реализующий метод Main() сервиса. Поскольку Service1 нельзя считать достаточно информативным именем для пользовательского сервиса, с помощью окна свойств укажите для свойств (Name) и ServiceName значение CarService. Различие между этими двумя свойствами в том, что значение (Name) задает имя, используемое для обращения к типу в программном коде, а свойство ServiceName обозначает имя, отображаемое в окне конфигурации сервисов Windows.

Перед тем как двигаться дальше, установите ссылки на компоновочные блоки CarGeneralAsm.dll и System.Remoting.dll, а также укажите следующие строки директив using в файле, содержащем определение класса CarService.

using System.Runtime.Remoting;

using System.Runtime.Remoting.Channels.Http;

using System.Runtime.Remoting.Channels;

using System.Diagnostics;

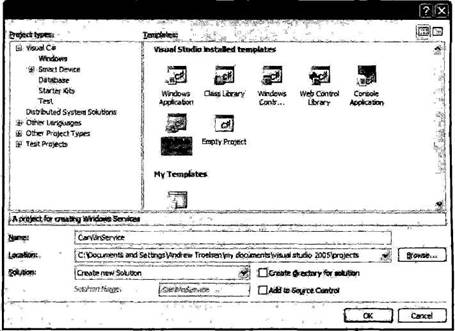


Рис. 18.7. Создание рабочего пространства нового проекта Windows Service

**Реализация метода Main()**

Метод Main() класса Program обеспечивает запуск сервисов, определённых в проекте, путём передачи массива типов ServiceBase статическому методу Service.Run(). При условии, что имя пользовательского сервиса было изменено с Service1 на CarService, вы должны иметь следующее определение класса (для ясности программного кода здесь удалены комментарии).

static class Program {

 static void Main() {

  ServiceBase[] ServicesToRun;

  ServicesToRun = new ServiceBase[] { **new CarService**() };

  ServiceBase.Run(ServicesToRun);

 }

}

**Реализация метода CarService.OnStart()**

Вы, вероятно, уже догадываетесь, какая программная логика должна использоваться при запуске пользовательского сервиса. Напомним, что целью CarService является выполнение той задачи, которую выполнял ваш консольный сервис. Поэтому, чтобы зарегистрировать CarService в виде WKO-синглета, доступного по протоколу HTTP, можете добавить в метод OnStart() следующий программный код (при использовании сервисов Windows для обслуживания удаленных объектов вместо "жестко" запрограммированной реализации можно использовать тип RemotingConfiguration, позволяющий загрузить файл \*.config удаленного взаимодействия на стороне сервера).

protected override void OnStart(string[] args) {

**// Создание нового HttpChannel.**

 HttpChannel с = new HttpChannel(32469);

 ChannelServices.RegisterChannel(c);

**// Регистрация WKO-типа одиночного вызова.**

 RemotingConfiguration.RegisterWellKnownServiceType(typeof(CarGeneralAsm.CarProvider), "CarProvider.soap", WellKnownObjectMode.SingleCall);

**// Сообщение об успешном старте.**

 EventLog.WriteEntry("CarWinService", "CarWinService стартовал успешно!", EventLogEntryType.Information);

}

Заметим, что после регистрации типа в журнал регистрации событий Windows (с помощью типа System.Diagnostics.EventLog) записывается пользовательское сообщение с информацией о том, что машина-хост успешно запустила ваш сервис.

**Реализация метода OnStop()**

Строго говоря, CarService не требует никакой программной логики остановки. Но для примера давайте направим в EventLog еще одно событие, на этот раз с информацией о завершении работы пользовательского сервиса Windows.

protected override void OnStop() {

 EventLog.WriteEntry("CarWinService", "CarWinService остановлен", EventLogEntryType.Information);

}

Теперь, когда сервис полностью оформлен, следующей задачей является установка этого сервиса на удаленной машине.

**Добавление установщика сервиса**

Чтобы получить возможность установки сервиса на выбранной вами машине, нужно добавить в текущий проект CarWinService дополнительный тип. Для установки любого сервиса Windows (созданного как средствами .NET, так и средствами Win32 API) требуется создание в реестре целого ряда записей, которые позволят ОС взаимодействовать с сервисом. Вместо создания этих записей вручную, можно просто добавить в проект сервиса Windows тип Installer (установщик), который правильно сконфигурирует тип, производный от ServiceBase, для установки на целевой машине.

Чтобы добавить установщик для CarService, откройте окно проектирования сервиса (с помощью двойного щелчка на файле CarService.cs в окне Solution Explorer), щелкните правой кнопкой в любом месте окна проектирования сервиса и выберите Add Installer (добавить установщик), рис. 18.8.

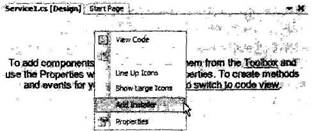


Рис. 18.8. Добавление установщика для пользовательского сервиса Windows

В результате будет добавлен новый компонент, который оказывается производным базового класса System.Configuration.Install.Installer. В окне проектирования теперь будет два компонента. Тип serviceInstaller1 представляет установщик конкретного сервиса в вашем проекте. Выбрав соответствующую пиктограмму, вы обнаружите, что в окне Properties для свойства ServiceName установлено значение типа CarService.

Второй компонент (serviceProcessInstaller1) позволяет задать окружение, в котором будет выполняться установленный сервис. По умолчанию значением свойства Асcount (учетная запись) является User (пользователь). С помощью окна свойств Visual Studio 2005 измените это значение на LocalServicе (локальный сервис), рис. 18.9.



Рис. 18.9. Идентификация СarService

Вот и все! Теперь скомпилируйте свой проект.

**Установка CarWinService**

Установка CarService.exe на машине (локальной или удаленной) предполагает выполнение двух действий.

1. Перемещение скомпилированного компоновочного блока сервиса (и всех необходимых внешних компоновочных блоков – в данном случае это CarGeneralAsm.dll) на удаленную машину.

2. Запуск средства командной строки installutil.exe с указанием соответствующего сервиса в качестве аргумента.

После выполнения п. 1 откройте командное окно Visual Studio 2005, перейдите в каталог с компоновочным блоком CarWinService.exe и введите следующую команду (этот же инструмент можно использовать и для деинсталляции сервиса).

installutil carwinservice.exe

После установки сервиса Windows вы можете запустить и сконфигурировать его с помощью апплета Services (Службы) Windows, который размещен в папке Администрирование панели управления Windows. В окне Services выделите CarService (рис. 18.10) и щелкните на ссылке Start (Запустить), чтобы загрузить и выполнить соответствующий двоичный файл.

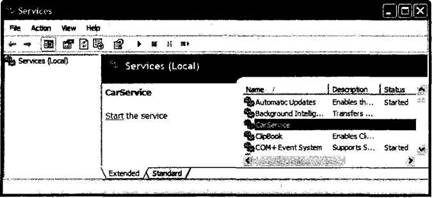


Рис. 18.10. Апплет Services Windows

**Исходный код.**Проект CarWinService размещен в подкаталоге, соответствующем главе 18.

**Хостинг удаленных объектов с помощью IIS**

Хостинг удаленного компоновочного блока с помощью сервера IIS (Internet Information Server – информационный сервер Интернет) даже проще, чем создание сервиса Windows, поскольку сервер IIS специально запрограммирован на то, чтобы получать поступающие запросы HTTP через порт 80. Поскольку IIS является Web-сервером, должно быть очевидно, что IIS может осуществлять обслуживание только удаленных объектов, использующих тип HttpChannel (в отличие от сервиса Windows, который допускает также использование типа TcpChannel). С учетом этого ограничения, при использовании IIS для поддержки удаленного взаимодействия необходимо выполнить следующие действия.

1. На жестком диске создайте новую папку для хранения CarGeneralAsm.dll. В этой папке создайте подкаталог \Bin. Скопируйте файл CarGeneralAsm.dll в этот подкаталог (например, C:\IISCarService\Bin).

2. На машине-хосте откройте окно апплета Internet Information Services (размещенного в папке Администрирование панели управления Windows).

3. Щелкните правой кнопкой в строке узла Default Web Site (Web-узел по умолчанию) и выберите New→Virtual Directory из появившегося контекстного меню.

4. Создайте виртуальный каталог, соответствующий только что созданной вами корневой папке (C:\IISCarService). Остальные значения, предложенные мастером создания виртуального каталога, будут вполне подходящими.

5. Наконец, создайте новый файл конфигураций с именем web.config для настройки параметров регистрации удаленных типов виртуальным каталогом (см. следующий фрагмент программного кода). Сохраните этот файл в соответствующей корневой папке (в данном случае это папка C:\IISCarService).

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

   ‹service›

    ‹wellknown mode="Singleton" type="CarGeneralAsm.CarProvider, CarGeneralAsm" objectUri="carprovider.soap" */›*

   ‹/service›

   ‹channels›

    ‹channel ref="http"/›

   ‹/channels›

  ‹/application*›*

 ‹/system.runtime.remoting›

‹/configuration›

Теперь файл CarGeneralAsm.dll будет доступен для НТТР-запросов IIS, и вы можете обновить файл \*.config на стороне клиента так, как показано ниже (конечно, указав в нем имя своего IIS-хоста).

‹configuration›

 ‹system.runtime.remoting›

  ‹application›

   ‹client displayName="CarClient"›

    ‹wellknown type="CarGeneralAsm.CarProvider, CarGeneralAsm" url=**"http://NameTheRemoteIISHost/IISCarHost/carprovider.soap**"/›

   ‹/client›

   ‹channels›

    ‹channel ref="http"/›

   ‹/channels›

  ‹/application›

 ‹/sуstem.runtime.remoting›

‹/configuration›

После этого вы сможете выполнять приложение клиента так же, как и раньше.

**Асинхронное удаленное взаимодействие**

В завершение нашего обсуждения материала данной главы давайте выясним, как вызывать члены удаленного типа асинхронно. В главе 14 была рассмотрена тема асинхронного вызова методов с помощью типов делегата. Как и следует ожидать, при асинхронном вызове удаленного объекта компоновочным блоком клиента первым шагом должно быть определение пользовательского делегата, представляющего соответствующий удаленный метод. После этого вызывающая сторона для вызова метода и получения возвращаемых значений может использовать любой из подходов, описанных в главе 14.

Для примера создайте новое консольное приложение (AsyncWKOCarProvider-Client) и установите в нем ссылку на первый вариант компоновочного блока CarGeneralAsm.dll. Теперь измените класс Program так, как показано ниже:

class Program {

**// Делегат для метода GetAllAutos().**

 internal delegate List‹JamesBondCar› GetAllAutosDelegate();

 static void Main(string[] args) {

  Console.WriteLine("Старт клиента! Для завершения нажмите ‹Enter›");

  RemotingConfiguration.Configure("AsyncWKOCarProviderClient.exe.config");

**// Создание поставщика машин.**

  CarProvider cp = new CarProvider();

**// Создание делегата.**

  GetAllAutosDelegate getCarsDel = new GetAllAutosDelegate(cp.GetAllAutos);

**// Асинхронный вызов GetAllAutos().**

  IAsyncResult ar = getCarsDel.BeginInvoke(null, null);

**// Имитация активности клиента.**

  while (!ar.IsCompleted) { Console.WriteLine("Клиент работает…"); }

**// Все сделано! Получение возвращаемого значения делегата.**

  List‹JamesBondCar allJBCs = getCarsDel.EndInvoke(ar);

**// Использование всех машин из списка.**

  foreach(JamesBondCar j in allJBCs) UseCar(j);

  Console.ReadLine();

 }

}

Здесь приложение клиента сначала объявляет делегат, соответствующий сигнатуре метода GetAllAutos() удаленного типа CarProvider. После создания делегата имя вызываемого метода (GetAllAutos) передается ему, как обычно. Потом запускается метод BeginInvoke(), сохраняется результирующий интерфейс IAsyncResult и имитируется какая-то работа на стороне клиента (напомним, что свойство IAsyncResult.IsCompleted позволяет выяснить, завершил ли работу соответствующий метод). После завершения работы клиента вы получаете список List‹›, возвращенный методом CarProvider.GetAllAutos() в результате вызова члена EndInvoke(), и передаете каждый объект JamesBondCar статической вспомогательной функции с именем UseCar().

public static void UseCar(JamesBondCar j) {

 Console.WriteLine("Может ли машина летать"? {0}", j.isFlightWorthy);

 Console.WriteLine("Может ли машина плавать? {0}", j.isSeaWorthy);

}

Снова подчеркнем, что красота использования типа делегата .NET заключается в том, что форма асинхронного вызова удаленных методов оказывается аналогичной форме вызова локальных методов.

**Исходный код**. Проект AsynсWKOCarProviderClient размещен в подкаталоге, соответствующем главе 18.

**Роль атрибута [OneWay]**

Предположим, что CarProvider должен иметь метод AddCar(), принимающий в качестве входного параметра JamesBondCar и не возвращающий ничего. Здесь главное те, что метод не возвращает ничего*.*Из названия класса System.Runtime. Remoting.Messaging.OneWayAttribute можно догадаться, что в данном случае слой удаленного взаимодействия .NET передает вызов удаленной стороне односторонним способом, и не заботится о создании инфраструктуры, необходимой для возврата значения (отсюда и название *one-way*- односторонний). Вот соответствующая модификация класса.

**// "Обитель" атрибута [OneWay].**

using System.Runtime.Remoting.Messaging;

…

namespace CarGeneralAsm {

 public class CarProvider: MarshalByRefObject {

  …

**// Клиент может вызвать соответствующий метод**

**// и 'забыть' о нем.**

  [OneWay]

  public void AddCar(JamesBondCar newJBC) { theJBCars.Add(newJBC); }

 }

}

Вызывающая сторона вызывает этот метод так, как обычно.

**// Создание поставщика машин.**

CarProvider ср = new CarProvider();

**// Добавление новой машины.**

ср.AddCar(new JamesBondCar("Zippy", 200, false, false));

С точки зрения клиента вызов AddCar() является полностью асинхронным, поскольку среда CLR обеспечивает использование фонового потока для запуска удаленного метода. Поскольку AddCar() сопровождается атрибутом [OneWay], клиент не может получить от вызова никакого возвращаемого значения. Но, поскольку AddCar() возвращает void, это не является проблемой.

Вдобавок к указанному ограничению, следует также знать о том, что при наличии у метода с атрибутом [OneWay] выходных или ссылочных параметров (определяемых с помощью ключевых слов out или ref) вызывающая сторона не сможет получить модификации вызываемой стороны. К тому же, если вдруг метод с атрибутом [OneWay] сгенерирует исключительное состояние (любого типа), вызывающая сторона ничего об этом не узнает. Удаленные объекты могут обозначить некоторые методы атрибутом [OneWay] только тогда, когда вызывающей стороне действительно позволяется вызвать эти методы и "забыть" об этом.

**Резюме**

В этой главе были рассмотрены варианты конфигурации компоновочных блоков .NET, позволяющие совместное использование типов за рамками одного приложения. Вы увидели, что удаленный объект можно сконфигурировать, как MBV-или MBR-тип. Именно от этого, в конечном счете, зависит то, как будет реализован удаленный тип в домене приложения клиента (в виде копии или с помощью прозрачного агента).

При конфигурации типа для работы в качестве MBR-объекта перед вами возникает целый ряд соответствующих вариантов выбора (WKO или САО, синглет или объект одиночного вызова и т.д.). Все имеющиеся варианты были рассмотрены в ходе обсуждения материала данной главы. Также были рассмотрены вопросы управления циклом существования удаленного объекта, реализуемого с помощью схемы лизингового управления и спонсорства лизинга. Наконец, снова была рассмотрена роль типов делегата .NET, используемых для асинхронного вызова удаленных методов (и здесь такой вызов по форме оказывается аналогичным асинхронному вызову методов локальных типов).

**ГЛАВА 19. Создание окон с помощью System.Windows.Forms**

Если вы прочитали предыдущие 18 глав, вы должны иметь солидную базу дли использования языка программирования C# и архитектуры .NET. Вы, конечно же, можете применить полученные знания для построения консольных приложений следующего поколения (как скучно!), но вас, наверное, больше интересует создание привлекательного графического интерфейса пользователя (GUI), который позволит пользователям взаимодействовать с вашей системой.

Эта глава является первой из трех глав, в которых обсуждается процесс построения традиционных приложений на основе использования так называемых форм. Вы узнаете, как создать "высокохудожественное" главное окно, используя классы Form и Application. В этой главе также показано, как в контексте GUI-окружения выполнить захват пользовательского ввода и ответить на него (т.е. обработать события мыши и клавиатуры). Наконец, вы узнаете, как вручную или с помощью инструментов проектирования, встроенных в Visual Studio 2005, конструировать системы меню, панели инструментов, строки состояния и интерфейс MDI (Multiple Document Interface – многодокументный интерфейс приложения).

**Обзор пространства имен System.Windows.Forms**

Как и любое другое пространство имен, System.Windows.Forms компонуется из различных классов, структур, делегатов, интерфейсов и перечней. Хотя различие между консольным (CUI) и графическим (GUI) интерфейсами, на первый взгляд, кажется подобным различию между ночью и днем, фактически для создания приложения Windows Forms необходимо только освоение правил манипуляции новым множеством типов с использованием того синтаксиса CIL, который вы уже знаете. С высокоуровневой точки зрения, сотни типов пространства имен System.Windows.Forms можно объединить в следующие большие категории.

*• Базовая инфраструктура.*Это типы, представляющие базовые операции программы .NET Forms (Form, Application и т.д.), а также различные типы, обеспечивающие совместимость с разработанными ранее элементами управления ActiveX.

• *Элементы управления.*Все типы, используемые для создания пользовательского интерфейса (Button, MenuStrip, ProgressBar, DataGridView и т.д.), являются производными базового класса Control. Элементы управления конфигурируются в режиме проектирования и оказываются видимыми (по умолчанию) во время выполнения.

• *Компоненты.*Это типы, не являющиеся производными базового класса Control, но тоже предлагающие визуальные инструменты (ToolTip, ErrorProvider и т.д.) для программ .NET Forms. Многие компоненты (например, Timer) во время выполнения невидимы, но они могут конфигурироваться визуально в режиме проектирования.

• *Диалоговые окна общего вида.*Среда Windows Forms предлагает целый ряд стандартных заготовок диалоговых окон для выполнения типичных действий (OpenFileDialog, PrintDialog и т.д.). Кроме того, вы можете создавать свои собственные пользовательские диалоговые окна, если стандартные диалоговые окна по какой-то причине вам не подойдут.

Поскольку общее число типов в System.Windows.Forms намного больше 100, кажется нерациональным (даже с точки зрения экономии бумаги) предлагать здесь описание всех элементов семейства Windows Forms. В табл. 19.1 описаны наиболее важные из типов System.Windows.Forms, предлагаемых в .NET 2.0 (все подробности можно найти в документации .NET Framework 2.0 SDK).

**Таблица 19.1.**Базовые типы пространства имен System.Windows.Forms

|  |  |
| --- | --- |
| **Классы** | **Описание** |
| Application | Класс, инкапсулирующий средства поддержки Windows Forms, необходимые любому приложению |
| Button, CheckBox, ComboBox, DateTimePicker, ListBox, LinkLabel, MaskedTextBox, MonthCalendar, PictureBox, TreeView | Классы, которые (вместе со многими другими классами) определяют различные GUI-элементы. Многие из этих элементов подробно будут рассмотрены в главе 21 |
| FlowLayoutPanel, TableLayoutPanel | Платформа .NET 2.0 предлагает целый набор "администраторов оформления", выполняющих автоматическую корректировку размещения элементов управления в форме при изменении ее размеров |
| Form | Тип, представляющий главное окно, диалоговое окно или дочернее окно MDI в приложении Windows Forms |
| ColorDialog, OpenFileDialog, SaveFileDialog, FontDialog, PrintPreviewDialog, FolderBrowserDialog | Представляют различные диалоговые окна, соответствующие стандартным операциям в рамках GUI |
| Menu, MainMenu, MenuItem, ContextMenu, MenuStrip, ContextMenuStrip | Типы, используемые для построения оконных и контекстно-зависимых систем меню. Новые (появившиеся в .NET 2.0) элементы управления MenuStrip и ContextMenuStrip позволяют строить меню, содержащие как традиционные пункты меню, так и другие элементы управления (окна текста, комбинированные окна и т.д.) |
| StatusBar, Splitter, ToolBar, ScrollBar, StatusStrip, ToolStrip | Типы, используемые для добавления в форму стандартных элементов управления |

**Замечание.**Вдобавок к System.Windows.Forms, компоновочный блок System.Windows. Forms.dll определяет и другие пространства имен, предназначенные для поддержки элементов графического интерфейса пользователя. Соответствующие дополнительные типы используются, в основном, внутренними механизмами создания форм и/или разработки Visual Studio 2005. По этой причине мы ограничимся рассмотрением базового пространства имен System.Windows.Forms.

**Работа с типами Windows Forms**

При построении приложения Windows Forms вы можете, при желании, создать весь соответствующий программный код вручную (например, в редакторе Блокнот или в редакторе TextPad), а затем отправить файлы *\*.*cs компилятору командной строки C# с флагом /target:winexe. Построение нескольких приложений Windows Forms вручную дает не только бесценный опыт, но и помогает лучше понять программный код, генерируемый графическими средствами проектирования, предлагаемыми в рамках пакетов интегрированной среды разработки .NET разных производителей.

Чтобы вы могли понять основы процесса создания приложений Windows Forms, в первых примерах этой главы не используются графические средства проектирования. Освоив процесс построений приложений Windows Forms "без помощи мастеров", вы без труда сможете перейти к использованию различных инструментов разработки, встроенных в Visual Studio 2005.

**Создание главного окна вручную**

В начале изучения приемов программирования Windows Forms мы построим самое простое главное окно, так сказать, "с чистого листа". Создайте на своем жест-ком диске новую папку (например, C:\MyFirstWindow) и в этой папке с помощью любого текстового редактора создайте новый файл MainWindow.cs.

В Windows Forms класс Form используется для представления любого окна в приложении. Это относится и к главному окну, находящемуся на вершине иерархии окон в приложении с интерфейсом SDI (Single-Document Interface – однодокументный интерфейс), и к модальным и немодальным диалоговым окнам, и к родительским и дочерним окнам в приложении с интерфейсом MDI (Multiple Document Interface – многодокументный интерфейс). Чтобы создать и отобразить главное окно приложения, необходимо выполнить следующие два обязательных шага.

1. Получить новый класс из System.Windows.Forms.Form.

2. Добавить в метод Main() приложения вызов метода Application.Run(), передав этому методу экземпляр производного от Form типа в виде аргумента.

Поэтому добавьте в файл MainWindow.cs следующее определение класса.

using System;

using System.Windows.Forms;

namespace MyWindowsApp {

 public class MainWindow : Form {

**// Выполнение приложения и идентификация главного окна.**

  static void Main(string[] args) {

   Application.Run(new MainWindow());

  }

 }

}

Вдобавок к обязательно присутствующему модулю mscorlib.dll, приложение Windows Forms должно сослаться на компоновочные блоки System.dll и System.Windows.Forms.dll. Вы, может быть, помните из главы 2, что используемый по умолчанию ответный файл C# (файл csc.rsp) дает указание csc.exe автоматически включить эти компоновочные блоки в процесс компиляции, так что здесь никаких проблем не ожидается. Также напомним, что опция /target:winexe компилятора csc.exe означает создание выполняемого файла Windows.

**Замечание.**Строго говоря, можно построить приложение Windows и с помощью опции /target:exe компилятора csc.exe, но тогда кроме главного окна полученное приложение в фоновом режиме будет создавать командное окно (которое будет существовать до тех пор, пока не завершит работу главное окно приложения). Указав /target:winexe, вы получите приложение, выполняемое в так называемом "родном" для Windows Forms режиме (без создания фонового командного окна).

Чтобы скомпилировать файл программного кода C#, откройте окно командной строки Visual Studio 2005 и выберите следующую команду.

csc /target:winexe \*.cs

На рис. 19.1 показан результат запуска полученного приложения.

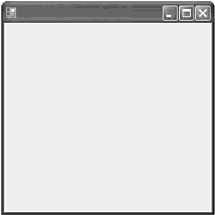


Рис. 19.1. Главное окно в стиле Windows Forms

Понятно, что такой результат применения средств Windows Forms впечатления не производит. Но обратите внимание на то, что путем получения простой производной от Form мы создали главное окно, допускающее минимизацию, максимизацию, изменение размеров и закрытие (да еще и с пиктограммой, предлагаемой системой по умолчанию!). В отличие от других средств разработки графического интерфейса от Microsoft, которые вы, возможно, использовали ранее (в частности, это касается библиотеки базовых классов MFC), теперь нет необходимости связывать сотни строк программного кода соответствующей инфраструктуры (фреймов, документов, представлений, приложений и карт сообщений). В отличие от приложений Win32 API, использующих C, здесь нет необходимости вручную реализовывать процедуры WinProc() и WinMain(). В рамках платформы .NET эту "грязную" работу выполняют элементы, инкапсулированные в типах Form и Application.

**Принцип разграничения обязанностей**

Сейчас класс MainWindow определяет метод Main() в рамках своего контекста. Но, если хотите, можно создать другой статический класс (назовем его Program). который будет отвечать за запуск главного окна, а задачей класса, производного от Form, останется непосредственное отображение окна.

namespace MyWindowsApp {

**// Главное окно.**

 public class MainWindow: Form {}

**// Объект приложения.**

 public static class Program {

  static void Main(string[] args) {

**// He забудьте о 'using' для System.Windows.Forms**!

   Application.Run(new MainWindow());

  }

 }

}

В результате вы обеспечите поддержку одного из главных правил объектно-ориентированного программирования – *разграничение обязанностей.*Это правило требует, чтобы класс был ответственен за выполнение минимально возможного объема работы. Разделив исходный класс на два отдельных класса, вы тем самым отделили форму от создающего ее класса. Результатом оказывается более мобильное (в смысле переносимости) окно, поскольку его теперь можно поместить в рамки любого проекта без дополнительного метода Main(), специфичного для данного проекта.

**Исходный код**. Проект MyFirstWindow размещён в подкаталоге, соответствующем главе 19.

**Роль класса Application**

Класс Application определяет множество статических членов, позволяющих управлять поведением различных низкоуровневых элементов приложения Windows Forms. Класс Application определяет набор событий, позволяющих реагировать, например, на завершение работы приложения или переход в состояние ожидания.

Кроме метода Run(), этот класс предлагает и другие методы, о которых вам следует знать.

• DoEvents(). Обеспечивает для приложения возможность в ходе выполнения операций, требующих много времени, обрабатывать сообщения, находящихся в это время в очереди сообщений.

• Exit(). Завершает выполнение Windows-приложения и выгружает из памяти домен этого приложения.

• EnableVisualStyles(). Настраивает приложение на поддержку визуальных стилей Windows XP. При активизации поддержки ХР-стилей указанный метод должен вызываться до загрузки главного окна с помощью Application.Run().

Кроме того, класс Application определяет ряд свойств, многие из которых доступны только для чтения. При анализе табл. 19.2 обратите внимание на то, что большинство свойств представляет характеристики "уровня приложения" (имя компании, номер версии и т.д.). С учетом ваших знаний об атрибутах уровня компоновочного блока (см. главу 12) многие из этих свойств должны быть для вас понятны.

**Таблица 19.2.**Основные свойства типа Application

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| CompanyName | Содержит значение атрибута [AssemblyCompany] уровня компоновочного блока |
| ExecutablePath | Содержит значение пути для файла, выполняемого в данный момент |
| ProductName | Содержит значение атрибута [AssemblyProduct] уровня компоновочного блока |
| ProductVersion | Содержит значение атрибута [AssemblyVersion] уровня компоновочного блока |
| StartupPath | Содержит значение пути для выполняемого файла, запустившего данное приложение |

Наконец, класс Application определяет набор статических событий, и вот некоторые из них.

• ApplicationExit генерируется непосредственно перед тем, как данное приложение завершает работу.

• Idle генерируется тогда, когда цикл сообщений приложения заканчивает обработку текущего пакета сообщений и готовится к переходу в состояние ожидания (ввиду отсутствия сообщений для обработки).

• ThreadExit генерируется непосредственно перед тем, как завершает работу поток данного приложения.

**Возможности класса Application**

Для иллюстраций некоторых функциональных возможностей класса Application давайте расширим тип MainWindow, чтобы он мог выполнять следующее.

• Использовать значения некоторых атрибутов уровня компоновочного блока.

• Обрабатывать статическое событие ApplicationExit…

Первой нашей задачей является использование свойств класса Application для отображения атрибутов уровня компоновочного блока. Для начала добавьте в свой файл MainWindow.cs следующие атрибуты (обратите внимание на то, что здесь используется пространство имен System.Reflection).

using System;

using System.Windows.Forms;

using System.Reflection;

**// Несколько атрибутов для данного компоновочного блока**.

[аssembly:AssemblyCompany("Intertech Training")] [assembly: AssemblyProduct("Более совершенное окно")"] [assembly:AssemblyVersion("1.1.0.0")]

namespace MyWindowsApp {

 …

}

Вместо того чтобы отображать атрибуты [AssemblyCompany] и [AssemblyProduct] вручную, используя приемы, предлагавшиеся в главе 12, класс Application позволяет сделать это автоматически, используя различные статические свойства. Например, можно реализовать конструктор следующего вида, который будет играть роль конструктора, заданного по умолчанию.

public class MainWindow: Form {

 publiс MainWindow() {

  MessageBox.Show(Application.**ProductName**, String.Format("Это приложение создано для вас компанией {0}", Application.**CompanyName**));

 }

}

Выполнив это приложение, вы увидите окно сообщения, отображающее соответствующую информацию (рис. 19.2).

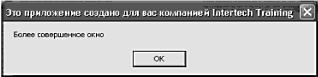


Рис 19.2. Чтение атрибутов с помощью типа Application

Теперь позволим форме отвечать на событие ApplicationExit. Вам, наверное, будет приятно узнать, что для обработки событий в рамках графического интерфейса приложений Windows Forms используется синтаксис событий, уже подробно описанный выше в главе 8. Поэтому, чтобы выполнить перехват статического события ApplicationExit, просто зарегистрируйте обработчик события с помощью операции +=.

public class MainForm: Form {

 public MainForm() {

**// Перехват события ApplicationExit.**

  Application.ApplicationExit += new EventHandler(MainWindow\_OnExit);

 }

 private void **MainWindow\_OnExit**(object sender, EventArgs evArgs) {

  MessageBox.Show(string.Format("Форма версии {0} завершила работу.", Application.ProductVersion));

 }

}

**Делегат System.EventHandler**

Обратите внимание на то, что событие ApplicationExit работает в паре с делегатом System.EventHandler. Этот делегат может указывать методы, соответствующие следующей сигнатуре.

delegate void EventHandler(object sender, EventArgs e);

Делегат System.EventHandler является самым примитивным делегатом, используемым для обработки событий Windows Forms, но существует очень много его вариаций. Что же касается EventHandler, то его первый параметр (типа System. Object) представляет объект, сгенерировавший данное событие. Второй параметр EventArgs (или его потомок) может содержать любую информацию, относящуюся к данному событию.

**Замечание.**Класс EventArgs является базовым для множества производных типов, содержащих дополнительную информацию для событий из определенных семейств. Так, для событий мыши используется параметр MouseEventArgs, предлагающий, например, такую информацию, как позиция (х, у) указателя. Для событий клавиатуры используется тип KeyEventArgs, предоставляющий информацию о текущих нажатиях клавиш и т.д.

Так или иначе, если вы перекомпилируете и запустите приложение, то теперь перед завершением работы приложения вы увидите соответствующий блок сообщения.

**Исходный код.**Проект AppClassExample размещен в подкаталоге, соответствующем главе 19.

**"Анатомия" формы**

Теперь, когда вы понимаете роль типа Application, следующей вашей задачей является непосредственное рассмотрение функциональных возможностей класса Form*.*Как и следует ожидать, класс Form наследует большинство своих функциональных возможностей от родительских классов. На рис. 19.3 показано окно Object Browser (в Visual Studio 2005), в котором отображается цепочка наследования производного от Form типа (вместе с набором реализованных интерфейсов).

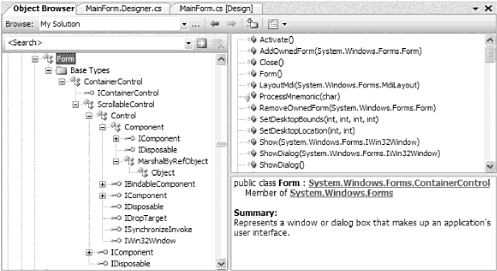


Рис. 19.3. Происхождение типа Form

Полная цепочка наследования типа Form включает в себя множество базовых классов и интерфейсов, но здесь следует подчеркнуть, что вам, чтобы стать хорошим разработчиком приложений Windows Forms, совеем *не обязательно*понимать роль каждого члена всех родительских классов и каждого реализованного интерфейса в этой цепочке. Значения большинства членов (в частности, большинство свойств и событий), которые вы будете использовать ежедневно, очень просто устанавливаются с помощью окна свойств Visual Studio 2005. Перед рассмотрением конкретных членов, унаследованных типом Form от родительских классов, изучите информацию табл. 19.3, в которой описана роль соответствующих базовых классов.

Вы, наверное, сами понимаете, что подробное описание каждого члена всех классов в цепочке наследования Form потребует отдельной большой книги. Важно понять общие характеристики поведения, предлагаемого типами Control и Form. Bсe необходимые подробности о соответствующих классах вы сможете найти в документации .NET Framework 2.0 SDK.

**Таблица 19.3.**Базовые классы из цепочки наследования Form

|  |  |
| --- | --- |
| **Родительский класс** | **Описание** |
| System.Object | Как и любой другой класс .NET, класс Form – это объект (Object) |
| System.MarshalByRefObject | При обсуждении возможностей удаленного взаимодействия .NET (см. главу 18) уже подчеркивалось, что типы, полученные из этого класса, будут доступны по *ссылке*(а не по копии) удаленного типа |
| System.ComponentModel.Component | Обеспечивает используемую по умолчанию реализацию интерфейса IComponent. В терминах .NET компонентом называется тип, поддерживающий редактирование в режиме проектирования, но не обязательно видимый во время выполнения |
| System.Windows.Forms.Control | Определяет общие члены пользовательского интерфейса для всех элементов управления Windows Forms, включая саму форму |
| System.Windows.Forms.ScrollableControl | Определяет автоматическую поддержку прокрутки содержимого |
| System.Windows.Forms.ContainerControl | Обеспечивает контроль фокуса ввода для тех элементов управления, которые могут выступать в качестве контейнера для других элементов управления |
| System.Windows.Forms.Form | Представляет любую пользовательскую форму, дочернее окно MDI или диалоговое окно |

**Функциональные возможности класса Control**

Класс System.Windows.Forms.Control задает общее поведение, ожидаемое от любого GUI-типа. Базовые члены Control позволяют указать размер и позицию элемента управления, выполнить захват событий клавиатуры и мыши, получить и установить фокус ввода, задать и изменить видимость членов и т.д. В табл. 19.4 определяются некоторые (но, конечно же, не все) свойства, сгруппированные по функциональности.

**Таблица 19.4.**Базовые свойства типа Control

|  |  |
| --- | --- |
| **Свойства** | **Описание** |
| BackColor, ForeColor, BackgroundImage, Font, Cursor | Определяют базовые параметры визуализации элемента управления (цвет, шрифт для текста, вид указателя мыши при его размещении на элементе и т.д.) |
| Anchor, Dock, AutoSize | Контролируют параметры размещения элемента управления в контейнере |
| Top, Left, Bottom, Right, Bounds, ClientRectangle, Height, Width | Указывают текущие размеры элемента управления |
| Enabled, Focused, Visible | Каждое из этих свойств возвращает значение типа Boolean, указывающее соответствующую характеристику состояния элемента управления |
| ModifierKeys | Статическое свойство, содержащее информацию о текущем состоянии модифицирующих клавиш (‹Shift›, ‹Ctrl› и ‹Alt›) и возвращающее эту информацию в вида типа Keys |
| MouseButtons | Статическое свойство, содержащее информацию о текущем состоянии кнопок мыши (левой, правой и средней) и возвращающее эту информацию в виде типа MouseButtons |
| TabIndex, TabStop | Используются для указания порядка переходов по клавише табуляции для элемента управления |
| Opacity | Определяет степень прозрачности элемента управления в дробных единицах (0.0 соответствует абсолютной прозрачности, а 1.0 – абсолютной непрозрачности) |
| Text | Указывает текстовые данные, ассоциируемые с элементом управления |
| Controls | Позволяет получить доступ к строго типизованной коллекции (ControlsCollection), содержащей все дочерние элементы управления, существующие в рамках данного элемента управления |

Кроме того, класс Control определяет ряд событий, позволяющих реагировать на изменение состояния мыши, клавиатуры, действия выделения и перетаскивания объектов (а также на многие другие действия). В табл. 19.5 предлагается описок некоторых (но далеко не всех) событий, сгруппированных по функциональности.

**Таблица 19.5.**События типа Control

|  |  |
| --- | --- |
| **События** | **Описание** |
| Click, DoubleClick, MouseEnter, MouseLeave, MouseDown, MouseUp, MouseMove, MouseHover, MouseWheel | События, позволяющие учитывать состояние мыши |
| KeyPress, KeyUp, KeyDown | События, позволяющие учитывать состояние клавиатуры |
| DragDrop, DragEnter, DragLeave, DragOver | События, используемые для контроля действий, связанных с перетаскиванием объектов |
| Paint | События, позволяющие взаимодействовать с GDI+ (см. главу 20) |

Наконец, базовый класс Control определяет целый ряд методов, позволяющих взаимодействовать с любым типом, производным от Control. При ближайшем рассмотрений методов Control вы обнаружите, что многие из них имеют префикс On, за которым следует имя соответствующего события (OnMouseMove, OnKeyUp, OnPaint и т.д.). Каждый из этих снабженных префиксом виртуальных методов представляет собой обработчик соответствующего события, заданный по умолчанию. Переопределив такой виртуальный член, вы получаете возможность выполнить необходимую предварительную (или заключительную) обработку данных, перед вызовом (или после вызова) родительской реализации обработчика события.

public class MainWindow: Form {

 protected **override**void OnMouseDown(MouseEventArgs e) {

**// Добавленный программный код для события MouseDown.**

**// Вызов родительской реализации.**

  base.OnMouseDown(e);

 }

}

Это может оказаться полезным, например, при создании пользовательских элементов управления, которые получаются из стандартных (см. главу 21), но чаще всего вы будете использовать обработку событий в рамках стандартного синтаксиса событий C# (именно это предлагается средствами проектирования Visual Studio 2005 по умолчанию). В этом случае среда разработки вызовет пользовательский обработчик события после завершения работы родительской реализации.

public class MainWindow: Form {

 public MainWindow() {

  MouseDown += new MouseEventHandler(MainWindow\_MouseDown);

 }

 void MainWindow\_MouseDown(object sender, MouseEventArgs e) {

**// Добавленный программный код для события MouseDown.**

 }

}

Кроме методов вида OnХХХ(), есть несколько других методов, о которые вам следует знать.

• Hide(). Скрывает элемент управления, устанавливая для его свойства Visible значение false (ложь).

• Show(). Делает элемент управления видимым, устанавливая для его свойства Visible значение true (истина).

• Invalidate(). Заставляет элемент управления обновить свое изображение, посылая событие Paint.

Несомненно, класс Control определяет и другие свойства, методы и события в дополнение к тем, которые вы только что рассмотрели. Но и сейчас вы должны иметь достаточно хорошее представление об общих функциональных возможностях этого базового класса. Давайте рассмотрим примеры, позволяющие увидеть указанный класс в действии.

**Использование возможностей класса Control**

Чтобы продемонстрировать возможности применения некоторых членов класса Control, давайте построим новую форму, способную обеспечивать следующее.

• Отвечать на события MouseMove и MouseDown.

• Выполнять захват и обработку ввода с клавиатуры, реагируя на событие KeyUp.

Для начала создайте новый класс, производный от Form. В конструкторе, заданном по умолчанию, мы используем различные наследуемые свойства, чтобы задать исходный вид и поведение формы. Обратите внимание на то, что здесь нужно указать использование пространства имён System.Drawing поскольку необходимо получить доступ к структуре Color (пространство имен System.Drawing будет рассмотрено в следующей главе).

using System;

using System.Windows.Forms;

using System.Drawing;

namespace MyWindowsApp {

 public class MainWindow: Form {

  publiс MainWindow() {

**// Использование наследуемых свойств для установки**

**// характеристик интерфейса пользователя.**

Text = "Моя фантастическая форма";

   Height = 300;

   Width = 500;

   BackColor = Color.LemonChiffon;

   Cursor = Cursors.Hand;

  }

 }

 public static class Program {

  static void Main(string[] args) {

   Application.Run(new MainWindow());

  }

 }

}

Скомпилируйте это приложение в его текущем виде, просто чтобы проверить что вы не допустили никаких опечаток.

csc /target:winexe \*.cs

**Ответ на события MouseMove**

Далее, мы должны обработать событие MouseMove. Целью является отображение текущих координат (x, у) указателя в области заголовка формы. Все связанные с состоянием мыши события (MouseMove. MouseUp и т.д.) работают в паре с делегатом MouseEventHandler, способным вызвать любой метод, соответствующий следующей сигнатуре.

void MyMouseHandler(object sender, MouseEventArgs e);

Поступающая на вход структура MouseEventArgs расширяет общий базовый класс EventArgs путем добавления целого ряда членов, специально предназначенных для обработки действий мыши (табл. 19.6).

**Таблица 19.6.**Свойства типа MouseEventArgs

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Button | Содержит информацию о том, какая клавиша мыши была нажата, в соответствии с определением перечня MouseButtons |
| Clicks | Содержит информацию о том, сколько раз была нажата и отпущена клавиша мыши |
| Delta | Содержит значение со знаком, соответствующее числу щелчков, произошедших при вращении колесика мыши |
| X | Содержит информацию о координате *х*указателя при щелчке мыши |
| Y | Содержит информацию о координате *у*указателя при щелчке мыши |

Вот обновленный класс MainForm, в котором обработка события MouseMove происходит так, как предполагается выше.

public class MainForm: Form {

 public MainForm() {

  …

**// Для обработки события MouseMove.**

  MouseMove += new MouseEventHandler(MainForm\_MouseMove);

 }

**// Обработчик события MouseMove.**

 public void MainForm\_MouseMove(object sender, MouseEventArgs e) {

  Text = string. Format ("Текущая позиция указателя: ({0}, {1})*",*е.Х, e.Y);

 }

}

Если теперь запустить программу и поместить указатель мыши на форму, вы увидите текущие значения координат (х, у) указателя, отображенные в области заголовка соответствующего окна (рис. 19.4).



Рис. 19.4. Мониторинг движения мыши

**Регистрация щелчков кнопок мыши**

Следует подчеркнуть, что событие MouseUp (как и MouseDown) посылается при щелчке *любой*кнопки мыши. Если нужно выяснить, какой кнопкой мыши был выполнен щелчок (левой, правой или средней), следует проанализировать значение свойства Button класса MouseEventArgs. Значение свойства Button соответствует одному из значений перечня MouseButtons. Предположим, что для обработки со-бытия MouseUp вы изменили заданный по умолчанию конструктор так, как показано ниже.

public MainWindow() {

 …

**// Для обработки события MouseUp.**

 MouseUp += new MouseHandler(MainForm\_MouseUp);

}

Следующий обработчик события MouseUp сообщает в окне сообщения о том, какой кнопкой мыши был выполнен щелчок.

public void MainForm\_MouseUp(object sender, MouseEventArgs e) {

**// Какая кнопка мыши была нажата?**

 if (e.Button == MouseButtons.**Left)**MessageBox.Show("Щелчок левой кнопки");

 if (e.Button == MouseButtons.**Right)**MessageBox.Show("Щелчок правой кнопки");

 if (e.Button == MouseButtons.**Middle)**MessageBox.Show("Щелчок средней кнопки");

}

**Ответ на события клавиатуры**

Обработка ввода с клавиатуры почти идентична обработке событий мыши. cобытия KeyUp и KeyDown работают в паре с делегатом KeyEventHandler, который может указывать на любой метод, получающий объект общего вида в качестве первого параметра, и KeyEventArgs – в качестве второго.

void MyKeyboardHandler(object sender, KeyEventArgs e);

Описания членов KeyEventArgs предлагаются в табл. 19.7.

**Таблица 19.7.**Свойства типа KeyEventArgs

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Alt | Содержит значение, являющееся индикатором нажатия клавиши ‹Alt› |
| Control | Содержит значение, являющееся индикатором нажатия клавиши ‹Ctrl› |
| Handled | Читает или устанавливает значение, являющееся индикатором полного завершения обработки события обработчиком |
| KeyCode | Возвращает клавишный код для события KeyDown или события KeyUp |
| Modifiers | Указывает, какие модифицирующие клавиши были нажаты (‹Ctrl›, ‹Shift› и/или ‹Alt›) |
| Shift | Содержит значение, являющееся индикатором нажатия клавиши ‹Shift› |

Измените объект MainForm, чтобы реализовать обработку события KeyUp. В окне сообщения отобразите название нажатой клавиши, используя свойство KeyCode.

public class MainForm: Form {

 public MainForm() {

  …

**// Для отслеживания событий KeyUp.**

  KeyUp += new KeyEventHandler(MainForm\_KeyUp);

 }

 private void MainForm\_KeyUp (object sender, KeyEventArgs e) {

  MessageBox.Show(e.**KeyCode.**ToString(), "Нажата клавиша!");

 }

}

Скомпилируйте и запустите программу. Теперь вы должны иметь возможность не только определить, какой кнопкой мыши был выполнен щелчок, но и то, какая была нажата клавиша на клавиатуре.

На этом мы завершим обсуждение функциональных возможностей базового класса Control и перейдем к обсуждению роли Form.

**Исходный код.**Проект ControlBehaviors размещен в подкаталоге, соответствующем главе 19.

**Функциональные возможности класса Form**

Класс Form обычно (но не обязательно) является непосредственным базовым классом для пользовательских типов Form. В дополнение к большому набору членов, унаследованных от классов Control, ScrollableControl и ContainerControl, тип Form предлагает свои собственные функциональные возможности, в частности для главных окон, дочерних окон MDI и диалоговых окон. Давайте сначала рассмотрим базовые свойства, представленные в табл. 19.8.

**Таблица 19.8.**Свойства типа Form

|  |  |
| --- | --- |
| **Свойства** | **Описание** |
| AcceptButton | Читает или устанавливает информацию о кнопке, которая будет "нажата" (в форме), когда пользователь нажмет клавишу ‹Enter› |
| ActiveMDIChild IsMDIChild IsMDIContainer | Используются в контексте МDI-приложения |
| CancelButton | Читает или устанавливает информацию о кнопочном элементе управления, который будет "нажат", когда пользователь нажмет клавишу ‹Esc› |
| ControlBox | Читает или устанавливает значение, являющееся индикатором наличия у формы экранной кнопки управления окном |
| FormBorderStyle | Читает или устанавливает значение, задающее стиль границы формы (в соответствии с перечнем FormBorderStyle) |
| Menu | Читает или устанавливает информацию о стыковке меню в форме |
| MaximizeBox MinimizeBox | Используются для информации о наличии у формы кнопок минимизации и максимизации окна |
| ShowInTaskbar | Указывает, будет ли форма видимой в панели задач Windows |
| StartPosition | Читает или устанавливает значение, задающее начальную позицию окна формы (в соответствии с перечнем FormStartPosition) |
| WindowState | Указывает (в соответствии с перечнем FormWindowState), в каком виде должна отображаться форма при запуске |

В дополнение к ожидаемым обработчикам событий с префиксом On, предлагаемым по умолчанию, в табл. 19.9 предлагается список некоторых базовых методов, определенных типом Form.

**Таблица 19.9.**Основные методы типа Form

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| Activate() | Активизирует форму и предоставляет ей фокус ввода |
| Close() | Закрывает форму |
| CenterToScreen() | Размещает форму в центре экрана |
| LayoutMDI | Размещает все дочерние формы (в соответствии с перечнем LayoutMDI) в рамках родительской формы |
| ShowDialog() | Отображает форму в виде модального диалогового окна. Более подробно о программировании диалоговых окон говорится в главе 21 |

Наконец, класс Form определяет ряд событий, связанных с циклом существования формы. Основные такие события описаны в табл. 19.10.

**Таблица 19.10.**Подборка событий типа Form

|  |  |
| --- | --- |
| **События** | **Описание** |
| Activated | Происходит при *активизации*формы, т.е. при получении формой фокуса ввода |
| Closed, Closing | Используются для проверки того, что форма закрывается или уже закрыта |
| Deactivate | Происходит при *деактивизации*формы, те. когда форма утрачивает текущий фокус ввода |
| Load | Происходит после того, как форма размещается в памяти, но пока остается невидимой на экране |
| MDIChildActive | Генерируется при активизации дочернего окна |

**Цикл существования типа Form**

Если вы имеете опыт программирования интерфейсов пользователя с помощью таких пакетов разработки, как Java Swing, Mac OS X Cocoa или Win32 АРI, вы должны знать, что "оконные типы" поддерживают множество событий, происходящих в различные моменты цикла существования таких типов. То же самое можно сказать и о типах Windows Forms. Вы уже видели, что "жизнь" формы начинается тогда, когда вызывается конструктор типа, перед его передачей методу Application.Run().

После размещения соответствующего объекта в управляемой динамической памяти среда разработки приложений генерирует событие Load. В обработчике событий Load можно настроить вид и поведение формы, подготовить содержащиеся в форме дочерние элементы управления (окна списков, деревья просмотра и т.д.), организовать доступ к ресурсам, необходимым для работы формы (установить связь с базами данных, создать агенты для удаленных объектов и т.д.).

Следующим событием, генерируемым после события Load, является событие Activated. Это событие генерируется тогда, когда форма получает фокус ввода, как активное окно на рабочем столе. Логическим "антиподом" события Activated является (конечно же) событие Deactivate, которое генерируется тогда, когда форма утрачивает фокус ввода, становясь неактивным окном. Легко догадаться, что события Activated и Deactivate в цикле существования формы могут генерироваться множество раз, поскольку пользователь может переходить от одного активного приложения к другому.

Когда пользователь решает закрыть соответствующую форму, по очереди генерируются еще два события: Closing и Closed. Событие Closing генерируется первым и дает возможность предложить конечному пользователю многими нелюбимое (но полезное) сообщение "Вы *уверены,*что хотите закрыть это приложение?". Этот шаг с требованием подтвердить выход полезен тем. что пользователю получает возможность сохранить данные соответствующего приложения перед завершением работы программы.

Событие Closing работает в паре с делегатом CancelEventHandler, определенным в пространстве имен System.ComponentModel. Если установить для свойства CancelEventArgs.Cancel значение true (истина), форме будет дано указание возвратиться к нормальной работе, и форма уничтожена не будет. Если установить для CancelEventArgs.Cancel значение false (ложь), будет сгенерировано событие Closed, и приложение Windows Forms будет завершено (домен приложения будет выгружен и соответствующий процесс прекращен).

Чтобы закрепить в памяти последовательность событий, происходящих в рамках цикла существовании формы, рассмотрим новый файл MainWindow.cs, в котором события Load, Activated, Deactivate, Closing и Closed обрабатываются в конструкторе класса так, как показано ниже (не забудьте добавить в программу директиву using для пространства имен System.ComponentModel, чтобы получить доступ к определению CancelEventArgs).

public MainForm() {

**// Обработка различных событий цикла существования формы.**

 Closing += new CancelEventHandler(MainForm\_Closing);

 Load += new EventHandler(MainForm\_Load);

 Closed += new EventHandler(MainForm\_Closed);

 Activated += new EventHandler(MainForm\_Activated);

 Deactivate += new EventHandler(MainForm\_Deactivate);

}

В обработчиках событий Load, Closed, Activated и Deactivate в строковую переменную System.String (с именем LifeTimeInfo) добавляется имя перехваченного события. Обработчик события Closed отображает значение этой строки в окне сообщения.

private void MainForm\_Load(object sender, System.EventArgs e) { lifeTimeInfo += "Событие Load\n"; }

private void MainForm\_Activated(object sender, System.EventArgs e) { lifeTimeInfo += "Событие Activate\n"; }

private void MainForm\_Deactivate(object sender, System.EventArgs e) { lifeTimeInfo += "Событие Deactivate\n"; }

private void MainForm\_Closed(object sender, System.EventArgs e) {

 lifeTimeInfo += "'Событие Closed\n";

 MessageBox.Show(lifeTimeInfо);

}

В обработчик события Closing задается вопрос о том, действительно ли пользователь желает завершить работу приложения. При этом используется поступающий на вход объект CancelEventArgs.

private void MainForm\_Closing(object sender, CancelEventArgs e) {

 DialogResult dr = MessageBox.Show("Вы ДЕЙСТВИТЕЛЬНО хотите закрыть приложение?", "Событие Closing", MessageBoxButtons.YesNo);

 if (dr == DialogResult.No) e.Cancel = true;

 else e.Cancel = false;

}

Обратите внимание на то, что метод MessageBox.Show() возвращает тип DialogResult, значение которого идентифицирует кнопку (Да, Нет), нажатую в форме конечным пользователем. Теперь скомпилируйте полученный программный код в командной строке.

csc /target:winexe \*.cs

Запустите приложение на выполнение и несколько раз поочередно предоставьте форме фокус ввода и уберите ее из фокуса ввода (чтобы сгенерировать события Activated и Deactivate). После прекращения работы вы увидите блок сообщений, аналогичный показанному на рис. 19.5.



Рис. 19.5. "Биография" типа, производного от Form

Большинство наиболее интересных возможностей типа Form связано с созданием и настройкой систем меню, панелей инструментов и строк состояния. Необходимый для этого программный код не слишком сложен, но Visual Studio 2005 предлагает целый набор графических инструментов проектирования, которые позаботятся о создании значительной части такого программного кода за вас. Поэтому давайте на время скажем "до свидания" компилятору командной строки и займемся созданием приложений Windows Forms с помощью Visual Studio 2005.

**Исходный код**. Проект FormLifeTime размещен в подкаталоге, соответствующем главе 19.

**Создание Windows-приложений в Visual Studio 2005**

В Visual Studio 2005 предлагается специальный шаблон для создания приложений Windows Forms. Выбрав шаблон Windows Application при создании проекта, вы получите не только объект приложения с соответствующим методом Main(), но и подходящий исходный тип, производный от Form. Кроме того, среда разработки предложит вам целый набор графических инструментов проектирования, способных превратить процесс построения интерфейса пользователя в детскую забаву. Чтобы рассмотреть имеющиеся возможности, создайте рабочее пространство нового проекта Windows Application (рис. 19.6). Мы пока что не собираемся создавать рабочий пример, так что можете назвать этот проект так, как захотите.

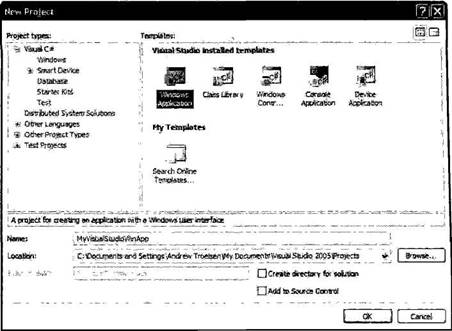


Рис. 19.6. Проект Windows Application в Visual Studio 2005

После загрузки проекта вы увидите окно проектирования формы, которое позволяет строить пользовательский интерфейс путем перетаскивания элементов управления и компонентов из панели инструментов (окно Toolbox, рис. 19.7), а также настраивать свойства и события этих элементов управления и компонентов с помощью окна свойств (окно Properties, рис. 19.8).



Рис. 19.7. Панель инструментов Visual Studio 2005

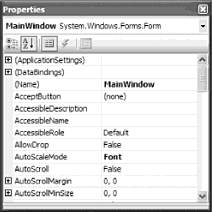


Рис. 19.8. Окно свойств Visual Studio 2005

Как видите, элементы управления в панели инструментов сгруппированы по категориям. В большинстве своем эти категории самоочевидны – например, категория Printing (Печать) содержит средства управления печатью. Menus & Toolbars (Меню и панели инструментов) содержит рекомендуемые элементы управления для меню и панелей инструментов и т.д. Тем не менее, пара категорий заслуживает специального обсуждения.

• Common Controls (Общие элементы управления). Элементы этой категории можно считать "рекомендуемым набором" общих элементов управления для построения пользовательского интерфейса.

• All Windows Forms (Все элементы управления Windows Forms). Здесь вы найдете полный набор элементов управления Windows Forms, включая элементы управления .NET 1.х, которые считаются устаревшими.

Второй из указанных здесь пунктов заслуживает более пристального внимания. Если вы работали с Windows Forms в рамках .NET 1.x, вам будет полезно знать, что многие привычные для вас элементы управления (например, элемент управления DataGrid) находятся как раз под "знаком" категории All Windows Forms. К тому же, некоторые общие элементы управления, которые вы могли использовать в рамках .NET 1.x (например, MainMenu, ToolBar и Statusbar) по умолчанию в панели Toolbox не показаны.

**Получение доступа к устаревшим элементам управления**

Во-первых, отметим, что устаревшие элементы пользовательского интерфейса, о которых здесь идет речь, остаются пригодными для использования в .NET 2.0, а во-вторых, если вы хотите их использовать, то их можно снова добавить в панель инструментов. Для этого щелкните правой кнопкой мыши в любом месте окна Toolbox (кроме строки заголовка) и выберите Choose Items (Выбрать элементы) из появившегося контекстного меню. В появившемся диалоговом после этого окне отметьте нужные вам элементы (рис. 19.9).

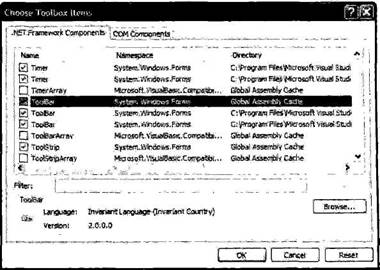


Рис. 19.9. Добавление элементов управления на панель инструментов

**Замечание**. Может показаться, что а списке в окне добавления элементов управления имеются повторения (например, для элемента управления ToolBar). На самом же деле каждый элемент списка уникален, так как соответствующий элемент управления может иметь другую версию (например, 2.0 вместо 1.0) и/или быть элементом .NET Compact Framework. Поэтому будьте внимательны, чтобы выбрать правильный элемент.

В этот момент, я уверен, вы спросите, по какой причине многие из старых элементов скрыты от просмотра. Причина в том, что .NET 2.0 предлагает новый набор меню, панелей инструментов и средств отображения состояния, которым сегодня отдается предпочтение. Например, вместо устаревшего элемента управления MainMenu для создания меню предлагается использовать элемент управления MenuStrip, обеспечивающий ряд новых функциональных возможностей в дополнение к возможностям, предлагавшимся в рамках MainMenu.

**Замечание**. В этой главе мы используем новое, рекомендуемое сегодня множество элементов управления пользовательского интерфейса. Чтобы получить информацию, необходимую для работы с устаревшими типами MainMenu, Statusbar и другими аналогичными им типами, обратитесь к документации .NET Framework 2.0 SDK.

**Анализ проекта Windows Forms в Visual Studio 2005**

Любой тип Form проекта Windows Forms в Visual Studio 2005 представлен двумя связанными C#-файлами, в чем можно убедиться непосредственно, заглянув в окно Solution Explorer (рис. 19.10).

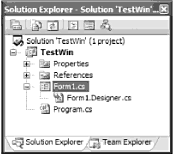


Рис. 19.10. Каждая форма является композицией двух файлов \*.cs

Щелкните правой кнопкой мыши на пиктограмме Form1.cs и в появившемся контекстном меню выберите View Code (Просмотр программного кода). Вы увидите программный код парциального класса, содержащего обработчики событий формы, конструкторы, переопределения и другие члены созданного вами класса (заметьте, что здесь имя исходного класса Form1 было изменено на MainWindow с помощью выбора Rename из меню Refactor).

namespace MyVisualStudioWinApp {

 public partial class MainWindow: Form {

  public MainWindow() {

   InitializeComponent();

  }

 }

}

Конструктор, заданный формой по умолчанию, вызывает метод InitializeComponent(), определенный в соответствующем файле \*.Designer.cs. Этот метод создается в Visual Studio 2005 автоматически, и в нем автоматически отражаются все модификации выполняемые вами в окне проектирования формы.

Для примера перейдите снова на вкладку окна проектирования формы и найдите свойство Text в окне свойств. Укажите для этого свойства новое значение (например, Мое тестовое окно). Теперь откройте файл Form1.Designer.cs и убедитесь в том, что метод InitializeComponent() соответствующим образом изменен.

private void InitializeComponent() {

 …

 this.Text = "Мое тестовое окно";

}

Кроме поддержки InitializeComponent(), файл \*.Designer.cs определяет члены-переменные, представляющие элементы управления, размещенные в окне проектирования формы. Снова для примера перетащите элемент управления Button (Кнопка) в окно проектирования формы. В окне свойств с помощью свойства Name измените имя соответствующей переменной с button1 на btnTestButton.

**Замечание**. Всегда лучше переименовать размещаемый элемент управления, перед тем как программировать обработку событий. Если этого не сделать, вы получите целый набор обработчиков событий с неинформативными именами наподобие button27\_Click, поскольку при создании соответствующего имени по умолчанию к имени элемента просто добавляется суффикс в виде порядкового номера переменной.

**Обработка событий в режиме проектирования**

Обратите внимание на то, что в окне свойств есть кнопка с изображением молнии. Вы, конечно, можете вручную создать программный код, обеспечивающий обработку событий уровня формы, (как это было сделано в предыдущих примерах), но эта кнопка позволяет обработать событие для данного элемента управления "визуально". Из раскрывающегося списка (вверху окна свойств) выберите элемент управления, который должен взаимoдeйствoвaть с формой, найдите событие, которое вы хотите обработать, и напечатайте имя, которое должно использоваться для обработчика события (или выполните двойной щелчок на имени события, чтобы сгенерировать типовое имя в виде ИмяЭлемента\_ИмяСобытия).

Если задать обработку события Click для элемента управления Button, в файле Form1.cs появится следующий обработчик событий.

public partial class MainWindow: Form {

 public MainWindow {

  InitializeComponent();

 }

 private void btnButtonTest\_Click(object sender, EventArgs e) {}

}

Файл Form1.Designer.cs будет содержать необходимую инфраструктуру и описание соответствующего члена-переменной.

partial class MainWindow {

 …

 private void InitializeComponent() {

  …

  this.btnButtonTest.Click += new System.EventHandler(this.btnButtonTest\_Click);

 }

 private System.Windows.Forms.Button btnButtonTest;

}

**Замечание**. Каждый элемент управлении имеет свое *событие по умолчанию,*которое будет обработано при двойном щелчке на этом элементе управления в окне проектирования формы. Например, событием по умолчанию для формы является Load, так что если выполнить двойной щелчок, поместив указатель мыши на тип Form среда разработки автоматически запишет программный код для обработки именно этого события.

**Класс Program**

Кроме файлов, связанных с формой, Windows-приложение Visual Studio 2005 определяет еще один класс, представляющий объект приложения (т.е. тип, определяющий метод Main()). Обратите внимание на то, что в следующем методе Main() вызывается Application.EnableVisualStyles(), а также Application.Run().

static class Program {

 [STAThread]

 static void Main() {

  Application.EnableVisualStyles();

  Application.Run(new MainWindow());

 }

}

**Замечание.**Атрибут [STAThread] дает среде CLR указание обрабатывать все устаревшие COM-объекты (включая элементы управления ActiveX), используя STA-управление (SingleThreaded Apartment – однопоточное размещение). Если вы имеете опыт использования COM, вы должны знать, что STA-управление используется для того, чтобы доступ к COM-типу выполнялся в синхронном (а значит, безопасном в отношении потоков) режиме.

**Необходимые компоновочные блоки**

Наконец, если Заглянуть в окно Solution Explorer, вы увидите, что проект Windows Forms автоматически ссылается на целый ряд компоновочных блоков, среди которых будут System.Windows.Forms.dll и System.Drawing.dll.

Напомним, что подробное обсуждение System.Drawing.dll предполагается в следующей главе.

**Работа с MenuStrip и ContextMenuStrip**

В рамках платформы .NET 2.0 рекомендуемым элементом управления для создания системы меню является MenuStrip. Этот элемент управления позволяет создавать как "обычные" пункты меню, такие как Файл→Выход, так и пункты меню, представляющие собой любые подходящие элементы управления. Вот некоторые общие элементы интерфейса, которые могут содержаться в MenuStrip.

• ToolStripMenuItem – традиционный пункт меню.

• ToolStripComboBox – встроенный элемент ComboBox (комбинированное окно).

• ToolStripSeparator – простая линия, разделяющая содержимое.

• ToolStripTextBox – встроенный элемент TextBox (текстовое окно).

С точки зрения программиста, элемент управления MenuStrip содержит строго типизированную коллекцию ToolStripItemCollection. Подобно другим типам коллекции, этот объект поддерживает методы Add(), AddRange(), Remove() и свойство Count. Эта коллекция обычно заполняется не напрямую, а с помощью различных инструментов режима проектирования, но если требуется, то есть возможность обработать ее и вручную.

Чтобы привести пример использования элемента управления MenuStrip, создайте новое приложение Windows Forms с именем MenuStripApp. Поместите элемент управления MenuStrip в форму в окне проектирования, присвоив ему имя mainMenuStrip. В результате в файл \*.Designer.cs добавится новая переменная.

private System.Windows.Forms.MenuStrip mainMenuStrip;

Элемент управления MenuStrip имеет очень широкие возможности настройки в режиме проектирования в Visual Studio 2005. Например, вверху у этого элемента управления есть маленькая пиктограмма стрелки, в результате выбора которой появляется контекстно-зависимый "встроенный" редактор содержимого, как показано на рис. 19.11.

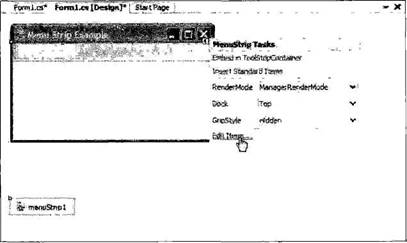


Рис. 19.11. "Встроенный" редактор MenuStrip

Такие контекстно-зависимые редакторы cодержимого поддерживаются многими элементами управления Windows Forms. Что же касается MenuStrip, то соответствующий редактор позволяет быстро сделать следующее.

• Вставить "стандартную" систему меню (File, Save, Tools, Help и т.д.), используя ссылку Insert Standard Items (Вставить стандартные элементы).

• Изменить стыковочное поведение MenuStrip.

• Отредактировать любой элемент MenuStrip (это просто "быстрая альтернатива" в отношении возможности выбора соответствующего конкретного элемента в окне свойств).

В этом примере мы проигнорируем возможности "встроенного" редактора, сосредоточившись на создании системы меню "вручную". Сначала в режиме проектирования выберите элемент управления MenuStrip и определите стандартное меню Файл→Выход, впечатав соответствующие имена в поле с подсказкой Type Here (Печатать здесь), рис. 19.12.

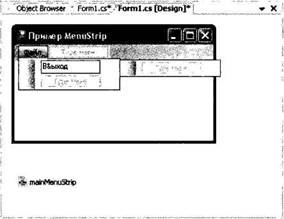


Рис. 19.12. Создание системы меню

**Замечание.**Вы, наверное, знаете, что символ амперсанда (&), размещенный перед буквой в строке элемента меню, задает комбинацию клавиш для быстрого вызова данного элемента. В этом примере указано &Файл→В&ыход, поэтому пользователь может активизировать меню Выход, нажав сначала ‹Alt+ф›, а затем ‹ы›.

Каждый элемент меню, введенный вами в режиме проектирования, представляется типом класса ToolStripMenuItem. Открыв свой файл \*.Designer.cs, вы увидите там новые переменные для каждого из введенных элементов.

partial class MainWindow {

 private System.Windows.Forms.MenuStrip mainMenuStrip;

 private System.Windows.Forms.ToolStripMenuItem fileToolStripMenuItem;

 private System.Windows.Forms.ToolStripMenuItem exitToolStripMenuItem;

}

При использовании редактора меню метод InitializeComponent() соответственно обновляется. Для MenuStrip во внутреннюю коллекцию ToolStripItemCollection добавляется элемент, соответствующий новому пункту меню высшего уровня (fileToolStripMenuItem). Точно так же обновляется переменная fileToolStripMenuItem, для которой в ее коллекцию ToolStripItemCollection вставляется переменная exitToolStripMenuItem с помощью свойства DropDownItems.

private void InitializeComponent() {

 …

**//**

**// menuStrip1**

**//**

 this.menuStrip1.**Items.AddRange**(new System.Windows.Forms.ToolStripItem[] { this.fileToolStripMenuItem });

 …

**//**

**// fileToolStripMenuItem**

**//**

 this.fileToolStripMenuItem**.DropDownItems.AddRange**(new System.Windows.Forms.ToolStripItem[] { this.exitToolStripMenuItem });

 …

**//**

**// MainWindow**

**//**

 this.Controls.Add(this.menuStrip1);

}

Наконец (что не менее важно), элемент управления MenuStrip добавляется в коллекцию элементов управления формы. Эта коллекции будет рассматриваться подробно в главе 21, а здесь важно отметить, чего элемент управления может быть видимым во время выполнения только в том случае, когда этот элемент присутствует в указанной коллекции.

Чтобы завершить создание программного кода нашего примера, вернитесь в режим проектирования и выполните обработку события Сlick для пункта меню Выход, используя кнопку событий в окне свойств. В сгенерированном обработчике событии выполните вызов ApplicationExit.

private void exitToolStripMenuItem\_Click(object sender, EventArgs e) {

 Application.Exit();

}

Теперь вы можете скомпилировать и выполнить свою программу. Проверьте, что вы можете завершить работу приложения как с помощью выбора Файл→Выход из меню, так и с помощью нажатия ‹Аlt+ф›, а затем ‹ы› на клавиатуре.

**Добавление элемента Textbox в MenuStrip**

Давайте создадим новый элемент меню наивысшего уровня, присвоив этому элементу имя Изменение Цвета фона. Подчиненным элементом в этом меню будет не пункт меню, а элемент ToolStripTextBox (рис. 19.13). Добавив новый элемент управления, измените его имя на toolStripTextBoxColor с помощью окна свойств.

Нашей целью является возможность выбора пользователем цвета (красный, зелёный, розовый и т.д.). значение которого будет установлено для свойства BackColor формы. Сначала обработайте событие LostFocus для нового члена. ToolStripTextBox в рамках конструктора формы (это событие происходит тогда, когда TextBox в ToolStrip перестает быть активным элементом интерфейса).

public MainWindow() {

 …

 toolStripTextBoxColor.LostFocus += new EventHandler(toolStripTextBoxColor\_LostFocus);

}

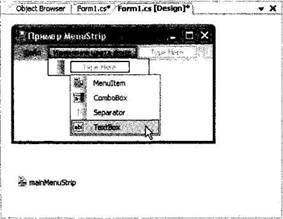


Рис. 19.13. Добавление TextBox в MenuStrip

В сгенерированном обработчике события прочитайте строковые данные, введенные в ToolStripTextBox (с помощью свойства Text), и используйте метод System. Drawing.Color.FromName(). Этот статический метод возвращает тип Color, соответствующий известному строковому значению. Чтобы учесть возможность ввода пользователем неизвестного цвета (или любых других неподходящих данных), используется простая логика try/catch.

void toolStripTextBoxColor\_LostFocus(object sender, EventArgs e) {

 try {

  BackColor = Color.FromName(toolStripTextBoxColor.Text);

 } catch {} **// Просто игнорировать неправильные данные.**

}

Запустите обновленное приложение снова и попробуйте ввести названия различных цветов. В результате правильного ввода вы должны увидеть изменение цвета фона формы. Чтобы получить информацию о допустимых названиях цветов, изучите информацию о типе System.Drawing.Color в окне обозревателя объектов (Object Browser) Visual Studio 2005 или в документации .NET Framework 2.0 SDK.

**Создание контекстных меню**

Рассмотрим теперь процедуру построения контекстно-зависимых меню (т.е. меню, раскрывающихся по щелчку правой кнопки мыши). Классом, используемым для построения контекстных меню в .NET 1.1. был класс ContextMenu, но в .NET 2.0 предпочтение отдается типу ContextMenuStrip. Подобно типу MenuStrip, тип ContextMenuStrip поддерживает ToolStripItemCollection для представления всех элементов меню (ToolStripMenuItem, ToolStripComboBox, ToolStripSeparator, ToolStripTextBox и т.д.).

Перетащите новый элемент управления ContextMenuStrip из панели инструментов в окно проектирования формы и измените имя этого элемента управления на fontSizeContextStrip с помощью окна свойств. Обратите внимание на то, что теперь дочерние элементы в ContextMenuStrip можно добавлять графически, почти так же, как при редактировании MenuStrip (очень приятное изменение по сравнению с методом, предлагавшимся в Visual Studio .NET 2003). Для нашего примера добавьте три элемента ToolStripMenuItem с названиями Крупный, Средний и Мелкий (рис. 19.14).

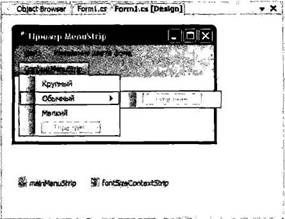


Рис. 19.14. Создание ContextMenuStrip

Это контекстное меню предназначено для того, чтобы пользователь мог выбрать размер шрифта для сообщения, отображаемого в области клиента формы. Чтобы упростить себе задачу, создайте тип перечня TextFontSize в рамках пространства имен MenuStripApp и объявите новый член-переменную этого типа в рамках Form (установив для переменной значение TextFontSize.FontSizeNormal).

namespace MainForm {

**// Вспомогательный перечень для размера шрифта.**

 enum TextFontSize {

  FontSizeHuge = 30,

  FontSizeNormal = 20,

  FontSizeTiny = 8

 }

 public class MainForm: Form {

**// Текущий размер шрифта.**

  private TextFontSize currFontSize = TextFontSize.FontSizeNormal;

  …

 }

}

Следующим шагом является обработка событий Paint формы с помощью окна свойств. Как будет показано в следующей главе, событие Paint позволяет отобразить в клиентской области формы графические данные (включая представленный в соответствующем стиле текст). В данном случае мы должны отобразить текстовое сообщение, используя указанный пользователем размер шрифта. Не беспокоясь пока что о деталях, модифицируйте обработчик события Paint так, как предлагается ниже.

private void MainWindow\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

 g.DrawString("Щелкните здесь правой кнопкой мыши…", new Font("Times New Roman", (float)currFontSize), new SolidBrush(Color.Black), 50, 50);

}

Наконец, нужно обработать события Click для каждого из типов ToolStripMenuItem, поддерживаемых элементом ContextMenuStrip. При этом можно, конечно, иметь отдельный обработчик событий Click для каждого из типов, но мы укажем один обработчик событий, который будет вызываться при щелчке на любом из трех элементов ToolStripMenuItem. Используя окно свойств, укажите для обработчика событий Click имя ContextMenuItemSelection\_Ciicked для всех трех типов ToolStripMenuItem и реализуйте соответствующий метод так, как показано ниже.

private void ContextMenuItemSelection\_Clicked(object sender, EventArgs e) {

**// Получение элемента ToolStripMenuItem,**

**// на котором выполнен щелчок.**

 ToolStripMenuItem miClicked = (ToolStripMenuItem)sender;

**// Поиск элемента, на которой выполнен щелчок, по его имени.**

 if (miClicked.Name == "hugeToolStripMenuItem") currFontSize = TextFontSize.FontSizeHuge;

 if (miClicked.Name == "normalToolStripMenuItem") currFontSize = TextFontSize.FontSizeNormal;

 if (miClicked.Name == "tinyToolStripMenuItem") currFontSize = TextFontSize.FontSizeTiny;

**// Указание форме обновить представление.**

 Invalidate();

}

Обратите внимание на то, что использование аргумента sender позволяет определить имя члена-переменной ToolStripMenuItem, чтобы установить размер текста. После этого вызов Invalidate() генерирует событие Paint, которое вызовет ваш обработчик события Paint.

Заключительным шагом является информирование формы о том, какой элемент ContextMenuStrip должен отображаться при щелчке правой кнопки мыши в области клиента. Для этого с помощью окна свойств установите значение свойства ContextMenuStrip равным имени элемента контекстного меню. После этого в контексте InitializeComponent() появится следующая строка.

this**.ContextMenuStrip**= this.fontSizeContextStrip;

Если выполнить приложение теперь, вы сможете изменить размер отображаемого текстового сообщения по щелчку правой кнопки мыши.

**Замечание.**С помощью свойства Context MenuStrip в контекстное меню можно включить любой элемент управления. Например, если в диалоговом окне контекстного меню создать объект Button (Кнопка), то соответствующий пункт Меню будет отображаться только тогда, когда щелчок будет выполнен в рабочей области кнопки.

**Проверка состояния элементов меню**

Члены типа ToolStripMenuItem позволяют проверить состояние элемента меню, сделать его доступным или скрытым. В табл. 19.11 даются описания некоторых из наиболее интересных свойств этого типа.

**Таблица 19.11.**Члены типа ToolStripMenuItem

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Checked | Получает или устанавливает значение, являющееся индикатором наличия отметки выбора в строке с текстом данного ToolStripMenuItem |
| CheckOnClick | Получает или устанавливает значение, являющееся индикатором необходимости появления отметки выбора для данного ToolStripMenuItem при щелчке |
| Enabled | Получает или устанавливает значение, являющееся индикатором доступности данного ToolStripMenuItem |

Давайте расширим ваше контекстное меню так, чтобы в нем рядом с выбранным в настоящий момент пунктом меню отображалась отметка выбора. Установить отметку для данного элемента меню очень просто (для этого нужно установить значение свойства Checked равным true). Однако для того, чтобы проследить, какой пункт меню должен быть отмечен, потребуется дополнительная программная логика. Одним из возможных подходов здесь является определение специальной переменной ToolStripMenuItem, которая будет представлять элемент, отмеченный в настоящий момент.

public class MainWindow: Form {

 …

**// Указывает отмеченный элемент.**

 private ToolStripMenuItem currentCheckedItem;

}

Напомним, что размером текста по умолчанию является TextFontSize.FontSizeNormal. С учетом этого начальным отмеченным элементам в ToolStripMenuItem должен быть normalToolStripMenuItem. Измените конструктор формы так, как показано ниже.

public MainWindow() {

**// Наследуемый метод для центрирования формы.**

 CenterToScreen();

 InitializeComponent();

**// Установка отметки выбора для элемента меню 'Средний'.**

 currentCheckedItem = normalToolStripMenuItem;

 currentCheckedItem.Checked = true;

}

Теперь вы имеете возможность программно идентифицировать отмеченный в настоящий момент элемент, и последним из необходимых шагов будет обновление обработчика события ContextMenuItemSelection\_Clicked(). В нем нужно снять отметку выбора с элемента, выбранного ранее, и отметить новый текущий объект ToolStripMenuItem в соответствии с новым выбором пользователя.

private void ContextMenuItemSelection\_Clicked(object sender, EventArgs e) {

**// Удаление отметки выбора для элемента.**

 currentCheckedItem.Checked = false;

 …

 if (miClicked.Name == "hugeToolStripMenuItem") {

  currFontSize = TextFontSize.FontSizeHuge;

  currentCheckedItem = hugeToolStripMenuItem;

 }

 if (miClicked.Name = "normalToolStripMenuItem") {

  currFontSize = TextFontSize.FontSizeNormal;

  currentCheckedItem = normalToolStripMenuItem;

 }

 if (miClicked.Name == "tinyToolStripMenuItem") {

  currFontSize = TextFontSize.FontSizeTiny;

  currentCheckedItem = tinyToolStripMenuItem;

 }

**// Установка отметки выбора для нового элемента.**

currentCheckedItem.Checked = true;

 …

}

На рис. 19.15 показан законченный проект MenuStripApp в действии.

**Исходный код.**Проект MenuStripApp размещен в подкаталоге, соответствующем главе 19.



Рис. 19.15. Установка и удаление отметок выбора для элементов ToolStripMenuItem

**Работа с StatusStrip**

В дополнение к системе меню многие формы предлагают поддержку*строки состояния,*которая обычно размещается в нижней части формы. Строка состояния может делиться на любое число "панелей" с текстовой (или графической) информацией, содержащей пояснения для пунктов меню, текущее время или специальные данные приложения.

Хотя поддержка строк состояния (с помощью типа System.Windows.Forms. StatusBar) предлагается с момента появления платформы .NET, в .NET 2.0 вместо простого элемента StatusBar предлагается использовать новый тип StatusStrip. Подобно обычной строке состояния, StatusStrip может состоять из любого числа панелей, содержащих текстовые/графические данные, предоставленные типом ToolStripStatus. Однако StatusStrip может содержать и дополнительные элементы, например, следующего вида.

• ToolStripProgressBar – встроенный индикатор выполнения (хода задания).

• ToolStripDropDownButton – встроенная кнопка, отображающая при щелчке на ней раскрывающийся список вариантов выбора.

• ToolStripSplitButton – подобен ToolStripDropDownButton, но отображает элементы раскрывающегося списка только тогда, когда пользователь щелкает непосредственно в области раскрывающегося списка. ToolStripSplitButton предлагает также поведение, аналогичное обычной кнопке, и поэтому может поддерживать обработку события Click.

Для примера мы построим новый объект MainWindow, в котором поддерживается простое меню (Файл→Выход и Справка→О программе) и StatusStrip. Левая панель строки состояния будет использоваться для отображения строковых данных, соответствующих выбранному в настоящий момент элементу меню (например, при выборе пользователем элемента Выход в строке будет отображаться "Выход из приложения").

Средняя часть строки состояния будет отображать одну из двух динамически создаваемых значений, соответствующих текущему времени и текущей дате. Наконец, правая часть строки состояния будет представлять тип ToolStripDropDownButton, позволяющий пользователю переключиться с отображения даты на отображение времени и наоборот (да еще и с пиктограммой счастливого лица в придачу!). На рис. 19.16 показано окно приложения в своем окончательном варианте.



Рис. 19.16. Приложение StatusStrip

**Создание системы меню**

Создайте новый проект приложения Windows Forms с именем StatusStripApp. Разместите элемент управления MenuStrip в окне проектирования формы и создайте два пункта меню (Файл→Выход и Справка→О программе). После этого задайте обработку событий Click (щелчок) и MouseHover (задержка указателя мыши) для каждого из дочерних элементов меню (Выход и О программе) с помощью окна свойств.

Реализация обработчика событий Click для элемента Файл→Выход просто завершает работу приложения, а обработчик событий Click для Справка→О программе отображает окно сообщения MessageBox.

private void exitToolStripMenuItem\_Click(object sender, EventArgs e) { Application.Exit(); }

private void aboutToolStripMenuItem\_Click(object sender, EventArgs e) { MessageBox.Show("My StatusStripApp!"); }

Обработчики событий MouseHover, отображающие подходящие подсказки в левой панели StatusStrip, мы с вами обновим немного позже. Пока что оставьте их пустыми.

**Настройка StatusStrip**

Добавьте в окно проектирования формы элемент управления StatusStrip и поменяйте имя этого элемента управления на mainStatusStrip. Следует понимать, что по умолчанию StatusStrip не содержит вообще никаких панелей. Для добавления трех панелей можно использовать разные подходы.

• Создать необходимый программный код вручную, без помощи инструментов проектирования (возможно, с помощью вспомогательного метода CreateStatusStrip(), вызываемого в рамках конструктора формы).

• Добавить нужные элементы в диалоговом окне, появляющемся при выборе ссылки Edit Items (Редактирование элементов) из меню контекстного редактора StatusStrip (см. рис. 19.17).

• Добавить нужные элементы по одному с помощью раскрывающегося меню новых элементов StatusStrip (рис. 19.18).

Мы используем раскрывающееся меню новых элементов. С помощью этого меню добавьте два новых типа ToolStripStatusLabel, назначив им имена toolStripStatusLabelMenuState и toolStripStatusLabelClock, и тип ToolStripDropDownButton с именем toolStripDropDownButtonDateTime. Как и следует ожидать, в результате этого в файл \*.Designer.cs будут добавлены новые члены-переменные и соответственно обновлен метод InitializeComponent().

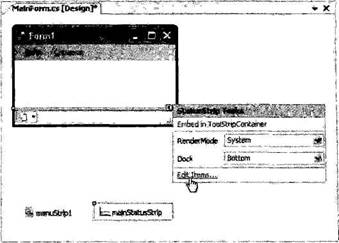
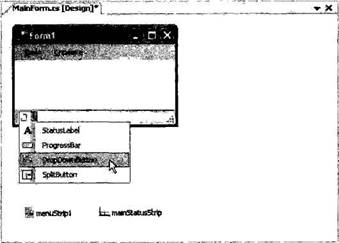


Рис. 19.17. Контекстный редактор StatusStrip



Риc. 19.18. Добавление элементов с помощью раскрывающегося меню новых элементов StatusStrip

Заметьте, что StatusStrip поддерживает внутреннюю коллекцию для представления всех созданных панелей.

partial class MainForm {

 private void InitializeComponent() {

  …

**//**

**// mainStatusStrip**

**//**

  this.mainStatusStrip.Items.AddRange(

   new System.Windows.Forms.ToolStripItem[] { this.toolStripStatusLabelMenuState, this.toolStripStatusLabelClock, this.toolStripDropDownButtonDateTime });

  …

 }

 private System.Windows.Forms.StatusStrip mainStatusStrip;

 private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabelMenuState;

 private System.Windows.Forms.ToolStripStatusLabel toolStripStatusLabelClock;

 private System.Windows.Forms.ToolStripDropDownButton toolStripDropDownButtonDateTime;

 …

}

Теперь в окне проектирования формы выберите ToolStripDropDownButton и добавьте два новых элемента меню День недели и Текущее время, соответственно назначив им имена dayoftheWeekToolStripMenuItem и currentTimeToolStripMenuItem (рис. 19.19).

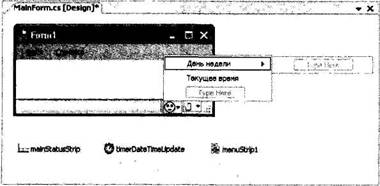


Рис. 19.19. Добавление пунктов меню для элемента ToolStripDropDownButton

Чтобы настроить панели так, как показано на рис. 19.19, нужно установить подходящие значения для соответствующих свойств в окне свойств Visual Studio 2005. В табл. 19.12 для элементов StatusStrip предлагаются описания свойств, которые нужно установить, и событий, которые нужно обработать (вы, конечно, можете настроить панели так, как сочтете необходимым).

Значение свойства Image члена toolStripDropDownButtonDateTime может указывать на любой файл с изображением, размещенный на вашей машине (при этом, конечно, следует учитывать то, что слишком большие файлы изображений могут порождать проблемы). Для нашего примера вы можете использовать файл happyDude.bmp, предлагаемый вместе с загружаемым исходным кодом для этой книги (посетите раздел загрузки Web-узла Apress, размещенный по адресу [http://www.apress.com](http://www.apress.com/)).

**Таблица 19.12.**Конфигурация панелей StatusStrip

|  |  |  |
| --- | --- | --- |
| **Член-переменная панели** | **Свойства для установки** | **События для обработки** |
| toolStripStatusLabelMenuState | Spring=true Text=(пусто) TextAlign=TopLeft | Нет |
| toolStripStatusLabelClock | BorderSides=All Text=(пусто) | Нет |
| toolStripDropDownButtonDateTime | Image=(см. ниже) | Нет |
| dayoftheWeekToolStripMenuItem | Text = "День недели" | MouseHover Click |
| currentTimeToolStripMenuItem | Text = "Текущее время" | MouseHover Click |

Итак, проектирование нашего графического интерфейса пользователя завершено. Но, чтобы реализовать оставшиеся обработчики событий, мы с вами должны выяснить роль компонента Timer (таймер).

**Работа с типом Timer**

Напомним, что средняя часть строки состояния должна отображать текущее время или текущую дату, в зависимости от предпочтений пользователя.

Первым шагом на пути к достижению этой цели является добавление в форму члена-переменной Timer – компонента, вызывающего некоторый метод (указанный с помощью обработчика события Tick) через заданный интервал времени (указанный с помощью свойства Interval).

Перетащите компонент Timer в окно проектирования формы и переименуйте его в timerDateTimeUpdate. Используя окно свойств, установите значение свойства Interval равным 1000 (это значение в миллисекундах), а значение свойства Enabled – равным true (истина). Наконец, обработайте событие Tick. Перед реализацией обработчика событий Tick определите в проекте новый тип перечня с именем DateTimeFormat. Этот перечень будет использоваться для выяснения того, что должен отображать второй элемент ToolStripStatusLabel – текущее время или текущую дату.

enum DateTimeFormat {

 ShowClock,

 ShowDay

}

Построив перечень, обновите MainWindow так, как предлагается ниже.

public partial class MainWindow: Form {

**// Какой формат отображать?**

 DateTimeFormat dtFormat = DateTimeFormat.ShowClock;

 …

 private void timerDateTimeUpdate\_Tick(object sender, EventArgs e) {

  string panelInfo = "";

**// Создание текущего формата.**

  if (dtFormat == DateTimeFormat.ShowClock) panelInfo = DateTime.Now.ToLongTimeString();

  else panelInfo = DateTime.Now.ToLongDateString();

**// Установка текста для панели.**

  toolStripStatusLabelClock.Text = panelInfo;

 }

}

Обратите внимание на то, что обработчик события Timer использует тип DateTime. Здесь вы просто читаете текущее время или дату системы, используя свойство Now, и устанавливаете соответствующее значение для свойства Text члена-переменной toolStripStatusLabelClock.

**Включение отображения**

В этот момент обработчик событий Tick должен отобразить в панели toolStripStatusLabelClock текущее время, если значением по умолчанию члена-переменной DateTimeFormat является DateTimeFormat.ShowClock. Чтобы позволить пользователю переключаться между отображением даты и времени, обновите MainWindow так, как предлагается ниже (заметьте, что здесь также указано, какой из двух пунктов меню в ToolStripDropDownButton должен при этом отмечаться).

public partial class MainWindow: Form {

**// Какой формат отображать?**

 DateTimeFormat dtFormat = DateTimeFormat.ShowClock;

**// Указывает отмеченный элемент.**

 private ToolStripMenuItem currentCheckedItem;

 public MainWindow() {

  InitializeComponent();

**// Эти свойства можно также установить**

**// в окне Properties.**

  Text = "Пример StatusStrip";

  CenterToScreen();

  BackColor = Color.CadetBlue;

  currentCheckedItem = currentTimeToolStripMenuItem;

  currentCheckedItem.Checked = true;

 }

 …

 private void currentTimeToolStripMenuItem\_Click(object sender, EventArgs e) {

**// Установка отметки и формата времени для панели.**

  currentCheckedItem.Checked = false;

  dtFormat = DateTimeFormat.ShowClock;

  currentCheckedItem = currentTimeToolStripMenuItem;

  currentCheckedItem.Checked = true;

 }

 private void dayoftheWeekToolStripMenuItem\_Click(object Sender, EventArgs e) {

**// Установка отметки и формата даты для панели.**

  currentCheckedItem.Checked = false;

  dtFormat = DateTimeFormat.ShowDay;

  currentCheckedItem = dayoftheWeekToolStripMenuItem;

  currentCheckedIteim.Checked = true;

 }

}

**Вывод подсказок для выбранных элементов меню**

Наконец, нужно настроить первую панель так. чтобы она содержала текст подсказки для выбранного пользователем элемента меню. Вы знаете, что большинство приложений отображает в левой части строки состояния поясняющую информацию (например, "Выход из приложения"), соответствующую выбранному конечным пользователем пункту меню. Если вы обработали события MouseHover для всех элементов меню нижнего уровня в MenuStrip и ToolStripDropDownButton, то остается только присвоить подходящее значение свойству Text для члена-переменной toolStripStatusLabelMenuState, например:

private void exitToolStripMenuItem\_MouseHover(object sender, EventArgs e) { toolStripStatusLabelMenuState.Text = "Выход из приложения"; }

private void aboutToolStripMenuItem\_MouseHover(object sender, EventArgs e) { toolStripStatusLabelMenuState.Text *=*"Отображение информации о приложении"; }

private void dayioftheWeekToolStripMenuItem\_MouseHover(object sender, EventArgs e) { toolStripStatusLabelMenuState.Text = "Отображение текущей даты."; }

private void currentTimeToolStripMenuItem\_MouseHover(object sender, EventArgs e) { toolStripStatusLabelMenuState.Text = "Отображение текущего времени."; }

Итак, у вас есть обновленный проект для тестового запуска. Теперь при выборе пунктов меню вы должны видеть в первой панели элемента StatusStrip соответствующие строки с поясняющей информацией.

**Состояние готовности**

Наконец, нужно гарантировать, что при снятии указателя мыши с пункта меню пользователем в первой текстовой панели не останется "старая" подсказка, а будет отображено некоторое "типовое" сообщение (например: "Ожидание действий пользователя"). В текущем своем виде наше приложение оставит в строке текст, соответствующий ранее выбранному пункту меню, что может вызывать, по меньшей мере, недоумение пользователя. Чтобы исправить это, обработайте событие MouseLeave для элементов меню Выход, О программе, День недели и Текущее время. Но вместо генерирования нового обработчика события для каждого элемента, позвольте всем указанным элементам вызывать один метод с именем SetReadyPrompt().

private void SetReadyPrompt(object sender, EventArgs e) { toolStripStatusLabelMenuState.Text = "Ожидание действий пользователя."; }

В результате вы должны обнаружить, что первая панель возвращается к отображению этого типового сообщения, как только курсор мыши покидает пределы любого из четырех указанных выше элементов меню.

**Исходный код.**Проект StatusBarApp размещен в подкаталоге, соответствующем главе 19.

**Работа с ToolStrip**

Тип ToolStrip в .NET2.0 предлагается использовать вместо типа ToolBar, предлагавшегося в рамках .NET 1.x и теперь считающегося устаревшим. Вы знаете, что панели инструментов обычно обеспечивают альтернативный способ активизации соответствующих пунктов меню. При щелчке пользователя на кнопке Сохранить, результат будет тем же, что и при выборе Файл→Сохранить из меню. Подобно MenuStrip и StatusStrip, тип ToolStrip может содержать множество разных элементов панели инструментов (возможности использования некоторых из них вы уже видели в предыдущих примерах).

• ToolStripButton

• ToolStripLabel

• ToolStripSplitButton

• ToplStripDropDownButton

• ToolStripSeparator

• ToolStripComboBox

• ToolStripTextBox

• ToolStripProgressBar

Подобно другим элементам управления Windows Forms, ToolStrip поддерживает встроенный редактор, который позволяет быстро добавить стандартные типы кнопок (File, Exit, Copy, Paste и т.д.), изменить поведение стыковки и встроить ToolStrip в ToolStripContainer (подробнее об этом чуть позже). Возможности поддержки ToolStrip в режиме проектирования демонстрируются на рис. 19.20.

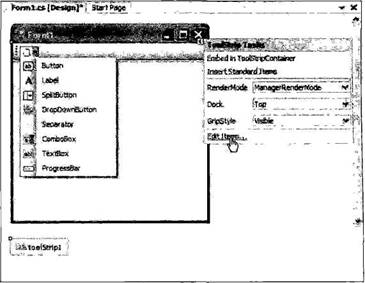


Рис. 19.20. Возможности режима проектирования для ToolStrip

Подобно MenuStrip и StatusStrip, индивидуальные элементы управления ToolStrip добавляются во внутреннюю коллекцию ToolStrip с помощью свойства Items (элементы). Если щелкнуть на ссылке Insert Standard Items (Вставить стандартные элементы) встроенного редактора ToolStrip, то в метод InitializeComponent() будет добавлен массив производных от ToolStripItem типов, представляющих соответствующие элементы.

private void InitializeComponent() {

 …

**// Автоматически генерируемый программный код**

**// для подготовки ToolStrip.**

 this.toolStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {

  this.newToolStripButton, this.openToolStripButton,

  this.saveToolStripButton, this.printToolStripButton,

  this.toolStripSeparator, this.cutToolStripButton,

  this.copyToolStripButton, this.pasteToolStripButton,

  this.toolStripSeparator, this.helpToolStripButton });

 …

}

Чтобы продемонстрировать работу с ToolStrip, в следующем приложении

Windows Forms создается тип ToolStrip, содержащий два типа ToolStripButton (с именами toolStripButtonGrowFont и toolStripButtonShrinkFont), тип ToolBarSeparator и тип ToolBarTextBox (с именем toolStripTextBoxMessage).

Конечный пользователь получает возможность ввести сообщение, которое будет отображено в окне формы с помощью ToolBarTextBox, а два типа ToolBarButton используются для того, чтобы увеличить или уменьшить размер шрифта. На рис. 19.21 показано результирующее окно проекта, который мы с вами собираемся построить.

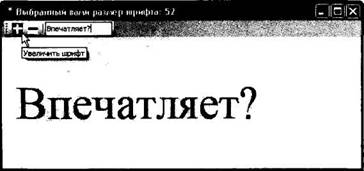


Рис. 19.21. Приложение ToolStripApp в действии

Я надеюсь, что к этому моменту вы имеете достаточно информации о приемах работы в режиме проектирования формы в Visual Studio 2005, поэтому я не буду утомлять вас указаниями по поводу построения ToolStrip. Однако следует заметить, что каждый элемент ToolStripButton имеет свою пользовательскую (хотя и достаточно примитивную) пиктограмму, которая была создана с помощью редактора изображений Visual Studio 2005. Если вы захотите создать файл изображения для своего проекта, можете просто выбрать Project→Add New Item из меню, а затем в появившемся диалоговом окне выбрать Icon File (Файл пиктограммы), рис. 19.22.

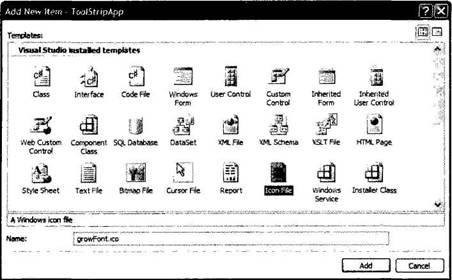


Рис. 19.22. Вставка новых файлов изображений

После этого вы можете отредактировать свои изображения с помощью окна Colors и комплекта инструментов редактора изображений. Имея пиктограммы в наличии, вы можете связать их с типами ToolStripButton с помощью свойства Image в окне свойств. После того как вы сочтете внешний вид ToolStrip удовлетворительным, обработайте событие Click для каждого элемента ToolStripButton.

Вот соответствующий программный код метода InitializeComponent() для первого типа ToolStripButton (второй ToolStripButton выглядит почти так же).

private void InitializeComponent() {

 …

**//**

**// toolStripButtonGrowFont**

**//**

 this.toolStripButtonGrowFont.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image;

 this.toolStripButtonGrowFont.Image = ((System.Drawing.Image)(resources.GetObject("toolStripButtonGrowFont.Image")));

 this.toolStripButtonGrowFont.ImageTransparentColor = System.Drawing.Color.Magenta;

 this.toolStripButtonGrowFont.Name = "toolStripButtonGrowFont";

 this.toolStripButtonGrowFont.Text = "toolStripButton2";

 this.toolStripButtonGrowFont.ToolTipText = "Увеличить шрифт";

 this.toolStripButton.GrowFont.Click += new System.EventHandler(this.toolStripButtonGrowFont\_Click);

 …

}

**Замечание**. Обратите внимание на то, что значение, присваиваемое свойству Image типа ToolStripButton, получается с помощью метода GetObject(). Как будет показано в следующей главе, этот метод используется для извлечения встроенных ресурсов, используемых компоновочным блоком.

Остальной программный код чрезвычайно прост. В следующем обновленном MainWindow обратите внимание на то, что текущий размер шрифта ограничивается значениями из диапазона между 12 и 70.

public partial class MainWindow: Form {

**// Текущий, минимальный и максимальный размеры шрифта.**

 int currFontSize = 12;

 const int MinFontSize =12;

 const int MaxFonfSize = 70;

 publiс MainWindow() {

  InitializeComponent();

  CenterToScreen();

  Text = string.Format("Выбранный вами размер шрифта: {0}", currFontSize);

 }

 private void toolStripButtonShrinkFont\_Click(object sender, EventArgs e) {

**// Уменьшение размера шрифта на 5 и обновление экрана.**

  currFontSize -= 5;

  if (currFontSize ‹= MinFontSize) currFontSize = MinFontSize;

  Text = string.Format("Выбранный вами размер шрифта: {0}", currFontSize);

  Invalidate();

 }

 private void toolStripButtonGrowFont\_Click(object sender, EventArgs e) {

**// Увеличение размера шрифта на 5 и обновление экрана.**

  currFontSize += 5;

  if (currFontSize ›= MaxFontSize) currFontSize = MaxFontSize;

  Text = string.Format("Выбранный вами размер шрифта: {0}", currFontSize);

  Invalidate();

 }

 private void MainWindow\_Paint(object sender, PaintEventArgs e) {

**// Отображение сообщения пользователя.**

  Graphics g = e.Graphics;

  g.DrawString(toolStripTextBoxMessage.Text, new Font("Times New Roman", currFontSize), Brushes.Black, 10, 60);

 }

}

В качестве заключительного штриха, чтобы гарантировать, что пользовательское сообщение будет обновлено, как только ToolStripTextBox утратит фокус, обработайте событие LostFocus и вызовите Invalidate() для формы в рамках сгенерированного обработчика события.

public partial class MainWindow: Form {

 …

 public MainWindow() {

  …

  this.toolStripTextBoxMessage.LostFocus += new EventHandler(toolStripTextBoxMessage\_LostFocus);

 }

 void toolStripTextBoxMessage\_LostFocus(object sender, EventArgs e) {

  Invalidate();

 }

 …

}

**Работа с ToolStripContainer**

Типы ToolStrip, если требуется, можно настроить так, чтобы они могли "стыковаться" с любой стороной и даже со всеми сторонами содержащей их формы. Для иллюстрации того, как это сделать, щелкните правой кнопкой мыши на своем элементе ToolStrip в окне проектирования фирмы и выберите Embed In ToolStripContainer (Встроить в контейнер). После этого ToolStrip будет помещен В ToolStripContainer. Для нашего примера выберите опцию Dock Fill In Form (Стыковка ко всей форме) риc. 10.23.

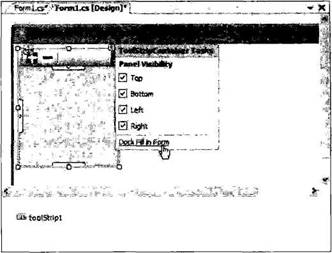


Рис. 19.23. Стыковка ToolStripContainer ко всей форме

Запустив эту модификацию приложения, вы увидите, что ToolStrip может перемещаться и стыковаться к любой стороне контейнера. Однако ваше пользовательское сообщение исчезнет. Причина в том, что типы ToolStripContainer являются для формы *дочерними элементами управления.* Так что графическое отображение на самом деле выполняется, но вывод скрыт контейнером, который сейчас накрывает всю клиентскую область формы.

Чтобы решить возникшую проблему, нужно обработать событие Paint для ToolStripContainer, а не для формы. Сначала найдите событие Paint формы в окне свойств и щелкните правой кнопкой на текущем обработчике событий. Из контекстного меню выберите Reset (рис. 19.24).

Это удалит программную логику обработки события из InitializeComponent(), но оставит программу обработки события на своем месте (просто, чтобы не допустить безвозвратную потерю программного кода, который вы, возможно, захотите использовать).

Теперь обработайте событие Paint для ToolStripContainer и переместите имеющийся программный код отображения из обработчика события Paint формы в обработчик события Paint контейнера. После этого можете удалить (теперь пустой) метод MainWindow\_Paint().

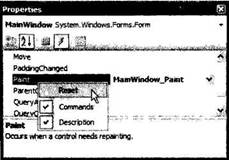


Рис. 19.24. Переустановка события

Наконец, нужно заменить каждый встречающийся вызов метода Invalidate() формы на вызов метода Invalidate() контейнера. Вот как выглядит соответствующая этому модификация программного кода.

public partial class MainWindow: Form {

 …

 void toolStripTextBoxMessage\_LostFocus(object sender, EventArgs e) {

**toolStripContainer1.Invalidate(true)**;

 }

 private void toolStripButtonShrinkFont\_Click(object sender, EventArgs e) {

  …

**toolStripContainer1.Invalidate(true);**

 }

 private void toolStripButtonGrowFont\_Click(object sender, EventArgs e) {

**toolStripContainer1.Invalidate(true);**

 }

**// Теперь "закрашивается" контейнер, а не форма!**

 private void ContentPanel\_Paint(object sender, PaintEventArgs e) {

  Graphics g = e.Graphics;

  g.DrawString(toolStripTextBoxMessage.Text, new Font("Times New Roman", currFontSize), Brushes.Black, 10, 60);

 }

}

Конечно, следует проверить разные конфигурации ToolStripContainer, чтобы понять, как все это работает. Подробности по документации .NET Framework 2.0 SDK вам придется изучать самостоятельно. На рис. 19.25 показано окно завершенного проекта.

**Исходный код**. Проект ToolStripApp размещен в подкаталоге, соответствующем главе 19.



Рис. 19.25. Приложение ToolStripApp с допускающим стыковку ToolStrip

**Создание MDI-приложения**

Чтобы завершить краткое знакомство с Windows Forms, давайте обсудим то, как настроить форму на работу в качестве родительского объекта для любого числа дочерних окон (т.е. в качестве MDI-контейнера). MDI-приложения дают пользователям возможность открывать множество дочерних окон в рамках одного и того же главного окна. В мире MDI каждое окно представляет свой "документ" приложения. Например, Visual Studio 2005 является МDI-приложением, поскольку вы можете открыть множество документов в рамках одного экземпляра этого приложения.

При построении МDI-приложения с помощью Windows Forms первой задачей оказывается (конечно же) создание нового приложения Windows. Исходная форма в приложении обычно содержит систему меню, которая позволяет создавать новые документы (например, содержит пункт Файл→Создать) и упорядочивать существующие открытые окна (каскадом, по вертикали или по горизонтали).

Для создания дочерних окон обычно определяется форма-прототип, играющая роль основы для каждого дочернего окна. Поскольку Form является типом класса, любые приватные данные, определенные в дочерней форме, будут для конкретного экземпляра уникальными. Например, при созданий MDI-приложения текстового редактора для отображения текста можно создать дочернюю форму, поддерживающую StringBuilder. Если пользователь создаст пять новых дочерних окон, каждая соответствующая форма будет поддерживать свой собственный экземпляр StringBuilder, которыми можно будет управлять независимо.

Кроме того, MDI-приложения позволяют "объединять" меню. Как уже упоминалось, родительские окна обычно имеют свои системы меню, которые позволяют пользователю создавать и упорядочивать дополнительные дочерние окна. Но что будет в том случае, когда дочернее окно имеет свою систему меню? Если пользователь максимизирует конкретное дочернее окно, то система меню этого дочернего окна должна "поглотиться" родительской формой, чтобы пользователь получил возможность активизировать элементы каждой из имеющихся систем меню. Пространство имен Windows Forms определяет ряд свойств, методов и событий, позволяющих программное слияние систем меню. Имеется также система "слияния по умолчанию", которая оказывается вполне подходящей во многих типичных случаях.

**Создание родительской формы**

Для демонстрации основ процесса построения MDI-приложения создайте новое приложение Windows, назвав его SimpleMdiApp. При этом почти вся MDI-ин-фраструктура может быть назначена исходной форме с помощью различных инструментов проектирования. Сначала найдите свойство IsMdiContainer в окне свойств и установите его равным true (истина). В результате в окне проектирования формы область клиента изменится – теперь она будет визуально представлять контейнер дочернего окна.

Затем разместите в главной форме новый элемент управления MenuStrip. В этом меню укажите три элемента высшего уровня с названиями Файл, Окно и Упорядочить окна. Меню Файл содержит два подчиненных элемента с названиями Создать и Выход. Меню Окно не содержит никаких подчиненных элементов, потому что при создании пользователем дополнительных дочерних окон новые элементы предполагается добавлять программно. Наконец, меню Упорядочить окна определяет три подчиненных элемента с названиями Каскадом, По вертикали и По горизонтали.

После создания меню пользовательского интерфейса обработайте событие Click для пунктов меню Выход, Создать, Каскадом, По вертикали и По горизонтали (напомним, что меню Окно пока что не имеет никаких подчиненных элементов). Обработчик элемента Файл→Создать мы реализуем в следующем разделе главы, а сейчас рассмотрим программный код для остальных элементов меню.

**// Обработка события Файл | Выход и упорядочение дочерних окон.**

private void cascadeToolStripMenuItem\_Click(object sender, EventArgs e) {

 LayoutMdi**(MdiLayout.Cascade)**;

}

private void verticalToolStripMenuItem\_Click(object sender, EventArgs e) {

 LayoutMdi**(MdiLayout.TileVertical);**

}

private void horizontalToolStripMenuItem\_Click(object sender, EventArgs e) {

 LayoutMdi**(MdiLayout.TileHorizontal)**;

}

private void exitToolStripMenuItem\_Click (object sender, EventArgs e) {

 Application.Exit();

}

Наибольший интерес здесь представляет использование метода LayoutMdi() и соответствующего перечня MdiLayout. Программный код обработки выбора каждого из элементов меню должен быть вам понятен. При выборе элемента пользователем вы даете указание родительской форме выполнить автоматическое размещение всех дочерних окон.

Перед тем как перейти к обсуждению процесса создания дочерних форм, установите еще одно свойство MenuStrip. Свойство MdiWindowListItem используется доя того, чтобы выяснить, какой пункт меню наивысшего уровня должен использоваться для автоматического списка имен всех дочерних икон при соответствующем выборе из меню. Присвойте значение этого свойства члену-переменной windowToolStripMenuItem. По умолчанию для этого списка используется значение дочернего свойства Text с числовым суффиксом (т.е. Form1, Form2, Form3 и т.д.).

**Создание дочерней формы**

Теперь, когда у вас есть оболочка MDI-контейнера, нужно создать дополнительную форму, выполняющую роль прототипа для данного дочернего окна. Начните со вставки нового типа Form в имеющийся проект (используйте Project→Add Windows Form), присвойте этому типу имя ChidPrototypeForm и обработайте для него событие Сlick. В сгенерированном обработчике события путем случайного выбора установите цвет фона для области клиента. Кроме того, выведите "преобразованное в строку" значение Color (цвет) нового объекта в полосу заголовка дочернего окна. Следующая программная логика реализует поставленные задачи.

private void ChildPrototypeForm\_Click(object sender, EventArgs e)

**// Получение трех случайных чисел.**

 int r, g, b;

 Random ran = new Random();

 r = ran.Next(0, 255);

 g = ran.Next(0, 255);

 b = ran.Next(0, 255);

**// Создание цветового значения для фона.**

 Color currColor = Color.FromArgb(r, g, b);

 this.BackColor = currColor;

 this.Text = currColor.ToString();

}

**Создание дочерних окон**

Заключительным шагом должно быть создание подходящей реализации обработчика событий Файл→Создать родительской формы. Теперь, когда дочерняя форма определена, соответствующая программная логика оказывается очень простой: нужно создать и отобразить новый экземпляр типа ChildPrototypeForm. Кроме того, нужно установить значение свойства MdiParent дочерней формы, указывающее на содержащую ее форму (в данном случае это ваше главное окно). Вот как должны выглядеть соответствующие модификации программы.

private void newToolStripMenuItem\_Сlick(object sender, EventArgs e) {

**// Создание нового дочернего окна.**

 ChildPrototypeForm newChild = new ChildPrototypeForm();

**// Ссылка на родительскую форму для данного дочернего окна.**

 newChild.MdiParent **=**this;

**// Отображение новой формы*.***

 newChild**.**Show();

}

**Замечание.**Дочерняя форма имеет возможность использовать свойство MdiParent непосредственно, когда требуется выполнить какие-то действия (или организовать сообщение) с родительским окном.

При тестировании этого приложения начните с создания нескольких дочерних окон и, щелкнув на каждом из них, создайте уникальные цвета для их фона, Если теперь рассмотреть подчиненные элементы меню Окно, вы должны обнаружить, что там представлена и учтена каждая дочерняя форма. Точно так же с помощью элементов меню Упорядочить окна вы можете дать указание родительской форме разместить дочерние формы по вертикали, горизонтали или каскадом. На рис. 19.26 показано окно готового приложения.

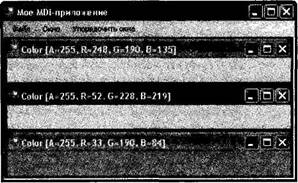


Рис. 19.26. Окно MDI-приложения

**Исходный код.**Проект SimpleMdiApp размещен в подкаталоге, соответствующем главе 19.

**Резюме**

Эта глава рассказывает об основах построения графического интерфейса с помощью типов, содержащихся в пространстве имен System.Windows.Forms. Сначала вам предлагается создать несколько приложений вручную, и в процессе этого выясняется, что GUI-приложение, как минимум, должно иметь класс, производный от Form, и метод Main(), вызывающий Application.Run().

В этой главе показано, как строить меню верхнего уровня (а также всплывающие меню) и как обрабатывать события меню. Было также выяснено, как можно расширить функциональные возможности типа Form с помощью панелей инструментов и строк состояния. В .NET 2.0 при создании таких элементов пользовательского интерфейса предлагается использовать MenuStrip, ToolStrip и StatusStrip, а не типы MainMenu, ToolBar и StatusBar из .NET 1.x (хотя эти, уже устаревшие типы, тоже поддерживаются). В завершение главы было продемонстрировано, как с помощью средств Windows Forms можно создавать MDI-приложения.

**ГЛАВА 20. Визуализация графических данных средствами GDI+**

Предыдущая глава предлагала вводное описание процесса построения GUI-приложений с помощью System.Windows.Forms. Целью этой главы является рассмотрение возможностей визуализации графических данных в окне формы (включая как вывод изображений, так и вывод текста различными стилями). Мы начнем с общего рассмотрения пространств имен, связанных с выводом графических данных, ради события Paint и "всемогущего" объекта Graphics.

Остальная часть этой главы будет посвящена способам манипуляции цветами, шрифтами, геометрическими формами и графическими образами. В этой главе также исследуется ряд программных подходов, связанных с визуализацией графических данных, например, таких как проверка доступа пользователя к непрямоугольным областям экрана, логика перетаскивания объектов и формат ресурсов .NET. И хотя, *строго говоря,*это не относится к GDI+ непосредственно, при операциях с ресурсами нередко приходится использовать манипуляции графическими данными (что оказывается достаточно важным для того, чтобы представить соответствующий материал здесь).

**Замечание.**Если вы программируете для Web, то можете подумать, что технологии GDI+ вам не пригодятся. Однако на самом деле эти технологии не ограничиваются только традиционными приложениями – они оказывается исключительно важными и для Web-приложений.

**Обзор пространств имен GDI+**

Платформа .NET обеспечивает целый набор пространств имен для поддержки визуализации двумерной графики. В дополнение к основным функциональным возможностям разработчика, которые обычно предлагаются графическими пакетами (цвета, шрифты, перья, кисти и т.д.), вы также найдете типы, осуществляющие геометрические трансформации, сглаживание, смешивание палитр и печать документов. Вместе эти пространства имен формируют тот набор возможностей .NET, который мы называем GDI*+*(Graphics Device Interface – интерфейс графических устройств, интерфейс GDI) и который является управляемой альтернативой Win32 GDI API (Application Programming Interface – программный интерфейс приложения). В табл. 20.1 предлагаются общие описания базовых пространств имен GDI+.

Таблица 20.1. Базовые пространства имен GDI+

|  |  |
| --- | --- |
| **Пространство имен** | **Описание** |
| System.Drawing | Базовое пространство имен GDI+, определяющее множество типов для основных операций визуализации (шрифты, перья, основные кисти и т.д.), а также "всемогущий" тип Graphics |
| System.Drawing.Drawing2D | Предлагает типы, используемые для более сложной двумерной/векторной графики (градиентные кисти, стили концов линий для перьев, геометрические трансформации и т.д.) |
| System.Drawing.Imaging | Предлагает типы, обеспечивающие обработку графических изображений (изменение палитры, извлечение метаданных изображения, работа с метафайлами и т.д.) |
| System.Drawing.Printing | Предлагает типы, обеспечивающие отображение графики на печатной странице, непосредственное взаимодействие с принтером и определение полного формата задания печати |
| System.Drawing.Text | Дает возможность управлять коллекциями шрифтов |

**Замечание.**Все пространства имен GDI+ определены в компоновочном блоке System.Drawing.dll. Многие типы проектов Visual Studio 2005 устанавливают ссылку на эту библиотеку программного кода автоматически, но вы можете при необходимости сослаться на System.Drawing.dll вручную, используя диалоговое окно Add References (Добавление ссылок).

**Обзор пространства имен System.Drawing**

Большинство типов, которые вам придется использовать при создании GDI-приложений, содержится в пространстве имен System.Drawing. Как и следует ожидать, здесь есть классы, представляющие изображения, кисти, перья и шрифты. Кроме того, System.Drawing определяет ряд связанных утилитарных типов, таких как Color (цвет), Point (точка) и Rectangle (прямоугольник). В табл. 20.2 предлагаются описания некоторых базовых типов этого пространства имен.

**Утилитарные типы System.Drawing**

Многие из методов визуализации, определенные объектом System.Drawing. Graphics, требуют указать позицию или область, в которой требуется отобразить данный элемент. Например, метод DrawString() требует, чтобы вы указали позицию, в которой нужно отобразить текстовую строку в производном от Control типе. Метод DrawString() является перегруженным, поэтому параметр позиции можно указать как в виде координаты (х, у), так и в виде размеров "бокса", в котором нужно выполнить визуализацию. Другие методы GDI+ могут требовать, чтобы вы указали ширину и высоту данного элемента или внутренние границы геометрического образа.

**Таблица 20.2.**Базовые типы пространства имен System.Drawing

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| Bitmap | Тип, инкапсулирующий данные изображения (\*.bmp или какого-то другого) |
| Brush Brushes SolidBrush SystemBrushes TextureBrush | Объекты Brush используются для заполнения внутренних областей графических форм, например, таких как прямоугольники, эллипсы и многоугольники |
| BufferedGraphics | Новый тип .NET 2.0, обеспечивающий графический буфер для двойной буферизации, которая используется для уменьшения или полного исключения влияния эффекта мелькания, возникающего при перерисовке изображений |
| Color SystemColors | Типы Color и SystemColors определяет ряд статических свойств, доступных только для чтения и используемых для получения нужного цвета при использовании различных перьев и кистей |
| Font FontFamily | Тип Font инкапсулирует характеристики данного шрифта (название, плотность, начертание, размер и т.д.)*.*FontFamily предлагает абстракцию для группы шрифтов, имеющих аналогичный дизайн, но определенные вариации стиля |
| Graphics | Представляет реальную поверхность нанесения изображения, а также предлагает ряд методов для визуализации текста, изображений и геометрических шаблонов |
| Icon SystemIcons | Представляют пользовательские пиктограммы, а также набор стандартных пиктограмм, предлагаемых системой |
| Image ImageAnimator | Тип Image – это абстрактный базовый класс, необходимый для поддержки функциональных возможностей типов Bitmap, Icon и Cursor. Тип ImageAnimator обеспечивает возможность выполнения цикла по набору типов Image из некоторого заданного интервала |
| Pen Pens SystemPens | Pens – это объекты, используемые для построения линий и кривых. Тип Pen определяет ряд статических свойств, возвращающих новый объект Pen заданного цвета |
| Point PointF | Структуры, представляющие отображение координаты (x, y) в соответствующее целое значение или значение с плавающей точкой, соответственно |
| Rectangle RectangleF | Структуры, представляющие размеры прямоугольника (снова с отображением в соответствующее целое значение или значение с плавающей точкой) |
| Size SizeF | Структуры, представляющие заданные высоту/ширину (снова с отображением в соответствующее целое значение или значение с плавающей точкой). |
| StringFormat | Тип, используемый для инкапсуляции различных характеристик размещения текста (выравнивание, промежутки между строками и т.д.) |
| Region | Тип, описывающий геометрический образ, скомпонованный из прямоугольников и траекторий |

Для указания такой информации пространство имен System.Drawing определяет типы Point, Rectangle, Region и Size. Очевидно, что тип Point (точка) представляет координату (x, у). Типы Rectangle (прямоугольник) содержат пару точек, представляющих левый верхний и нижний правый угол прямоугольной области. Типы Size (размер) подобны Rectangle, но эта структура представляет конкретные размеры, используя длину и ширину. Наконец, типы Region (регион) предлагают способ представления непрямоугольных областей.

Члены-переменные, используемые типами Point, Rectangle, Region и Size, внутренне представлены целочисленными данными. Если вам потребуется более "тонкая" детализация, можете использовать, соответственно, типы PointF, RectangleF и SizeF, которые (как вы можете догадаться) отображаются в соответствующие значения с плавающим разделителем. Но, независимо от внутреннего представления, все эти типы имеют аналогичные множества членов, включая ряд перегруженных операторов.

**Тип Point(F)**

Первым утилитарным типом, о котором вам следует знать, является тип System.Drawing.Point(F). В отличие от иллюстративных типов Point, создававшихся в предыдущих главах, тип Point(F) GDI+ поддерживает целый ряд очень полезных членов, включая следующие:

• +, -, ==, != – перегруженные варианты различных C#-операций;

• X, Y – обеспечивают доступ к соответствующим внутренним значениям (х, у) типа Point;

• IsEmpty – возвращает true (истина), если x и у установлены равными 0.

Для иллюстрации работы с утилитарными типами GDI+ рассмотрите следующее консольное приложение (названное UtilTypes), в котором используется тип System.Drawing.Point (не забудьте установить ссылку на System.Drawing.dll).

using System;

using System.Drawing;

namespace UtilTypes {

 public class Program {

  static void Main(string[] args) {

**// Создание и смещение точки.**

   Point pt = new Point(100, 72);

   Console.WriteLine(pt);

   pt.Offset(20, 20);

   Console.WriteLine(pt);

**// Перегруженные операции Point.**

   Point pt2 = pt;

   if (pt == pt2) WriteLine ("Точки одинаковы");

   else WriteLine("Точки различны");

**// Изменение значения X для pt2**.

   pt2.X = 4000;

**// Отображение каждого значения X.**

   Console.WriteLine("Первая точка: {0} ", pt);

   Console.WriteLine("Вторая точка: {0} ", рt2);

   Console.ReadLine();

  }

 }

}

**Тип Rectangle(F)**

Типы Rectangle, подобно Point, оказываются полезными во многих приложениях (и особенно в GUI-приложениях). Одним из наиболее полезных методов типа Rectangle является метод Contains(). Этот метод позволяет выяснить, находится ли данный тип Point или Rectangle в рамках границ некоторого другого объекта. Позже в этой же главе вы увидите, как использовать этот метод для проверки попадания в область GDI-изображений. А пока что рассмотрите следующий простой пример.

static void Main(string[] args) {

 …

**// Вначале Point находится вне прямоугольника**.

 Rectangle r1 = new Rectangle(0, 0, 100, 100);

 Point pt1 = new Point(101, 101);

 if (r1.**Contains(**pt3)) Console.WriteLine("Point находится внутри прямоугольника!");

 else Console.WriteLine("Point находится вне прямоугольника!");

**// Теперь поместим Point в прямоугольник.**

 pt3.X *=*50;

 pt3.Y = 30;

 if (r1.**Contains(**pt3)) Console.WriteLine("Point находится внутри прямоугольника!");

 else Console.WriteLine("Point находится вне прямоугольника!");

 Console.ReadLine();

}

**Класс Region**

Тип Region представляет внутреннюю часть геометрической фигуры. С учетом этого становится ясно, почему конструкторы класса Region требуют, чтобы вы предоставили им на вход некоторый уже существующий геометрический шаблон. Предположим, например, что вы создали прямоугольник размером 100×100 пикселей. Чтобы получить доступ к внутренней области прямоугольника, вы можете написать следующее.

**// Получение внутренней части прямоугольника.**

Rectangle r = new Rectangle(0, 0, 100, 100);

Region rgn = new Region(r);

Имея внутреннюю часть фигуры, вы можете манипулировать ею с использованием различных членов, наподобие следующих:

• Complement() – изменяет данный объект Region на часть указанного графического объекта, не пересекающуюся с данным объектом Region;

• Exclude() – изменяет данный объект Region на ту его часть, которая не пересекается с указанным графическим объектом;

• GetBounds() – возвращает Rectangle(F), который представляет прямоугольный регион, ограничивающий данный объект Region;

• Intersect() – изменяет данный объект Region на его пересечение с указанным графическим объектом:

• Transform() – трансформирует данный объект Region с помощью указанного объекта Matrix;

• Union() – изменяет данный объект Region на его объединение с указанным графическим объектом;

• Translate() – сдвигает координаты данного объекта Region на указанную величину.

Надеюсь, что вы получили общее представление об этих координатных примитивах. Если же вам нужны подробности, обратитесь к документации .NET Framework 2.0 SDK.

**Замечание.**Типы Size и SizeF заслуживают небольшого дополнительного комментария. Каждый из этих типов определяет свойства Height (высота) и Width (ширина), а также набор перегруженных операций.

**Исходный код.**Проект UtilTypes размещен в подкаталоге, соответствующем главе 20.

**Класс Graphics**

Класс System.Drawing.Graphics – это "вход" в функциональные возможности визуализации GDI+. Этот класс не только представляет поверхность, на которой вы хотите разместить изображение (например, поверхность формы, поверхность элемента управления или область в памяти), но определяет также десятки членов, которые позволяют отображать текст, изображения (пиктограммы, точечные рисунки и т.д.) и самые разные геометрические формы. Частичный список членов данного класса представлен в табл. 20.3.

Кроме ряда методов визуализации, класс Graphics определяет дополнительные члены, позволяющие конфигурировать "состояние" объекта Graphics. С помощью присвоения подходящих значений свойствам, показанным в табл. 20.4, вы можете изменить текущие характеристики процесса визуализации.

**Таблица 20.3.**Члены класса Graphics

|  |  |
| --- | --- |
| **Методы** | **Описание** |
| FromHdc() FromHwnd() FromImage() | Статические методы, обеспечивающие возможность  получения действительного объекта Graphics из  данного изображения (например, пиктограммы,  точечного рисунка и т.п.) или GUI-элемента |
| Clear() | Заполняет объект Graphics заданным цветом,  выполняя в процессе заполнения очистку  поверхности рисования |
| DrawArc() DrawBezier() DrawBeziers()  DrawCurve() DrawEllipse() DrawIcon()  DrawLine() DrawLines() DrawPath()  DrawRectangle() DrawRectangles()  DrawString() | Эти методы используются для визуализации  данного изображения или геометрического шаблона  . Позже вы увидите, что методы DrawXXX() требуют  использования объектов Pen GDI+ |
| FillEllipse() FillPath() FillPie() FillPolygon() FillRectangle() | Эти методы иcпользуются для заполнения внутренности  данной геометрической формы. Позже вы увидите,  что методы DrawXXX() требуют использования  объектов Brush GDI+ |

**Таблица 20.4.**Свойства класса Graphics, сохраняющие состояние

|  |  |
| --- | --- |
| **Свойства** | **Описание** |
| Clip ClipBounds VisibleClipBounds  IsClipEmpty IsVisibleClipEmpty | Позволяют установить опции отсечения, используемые с текущим  объектом Graphics |
| Transform | Позволяет трансформировать "мировые координаты" (подробнее  об этом будет говориться позже) |
| PageUnit PageScale DpiX DpiY | Позволяют указать начало координат для операций визуализации,  а также единицу измерения |
| SmoothingMode PixelOffsetMode  TextRenderingHint | Позволяют задать параметры гладкости геометрических объектов  и текста |
| CompositingMode CompositingQuality | Свойство CompositingMode задает режим визуализации: либо  рисование поверх фона, либо сопряжение с фоном |
| InterpolationMode | Указывает режим интерполяции данных между конечными точками |

**Замечание.**В .NET 2.0 пространство имен System.Drawing предлагает тип BufferedGraphics, который позволяет отображать графику, используя систему двойной буферизации, чтобы ослабить или исключить возможное мерцание, происходящее при визуализации данных. Подробная информация об этом есть в документации .NET Framework 2.0 SDK.

Обращаем ваше внимание на то, что класс Graphics не допускает непосредственного создания своего экземпляра с помощью ключевого слова new, поскольку этот класс не имеет открытых конструкторов. Но тогда как получить объект Graphics? Я рад, что вы спросили об этом.

**Сеансы Paint**

Наиболее общий способ получения объекта Graphics заключается во взаимодействии с событием Paint. Вспомните из предыдущей главы о том, что класс Control определяет виртуальный метод с именем OnPaint(). Чтобы форма отображала графические данные на своей поверхности, вы можете переопределить этот метод и извлечь объект Graphics из входного параметра PaintEventArgs. Для примера создайте новое приложение Windows Forms с именем BasicPaintForm и обновите полученный класс Form так, как предлагается ниже.

public partial class MainForm: Form {

 public MainForm() {

  InitializeComponent();

  CenterToScreen();

  this.Text = "Basic Paint Form";

 }

 protected override void OnPaint(PaintEventArgs e) {

**// При переопределении OnPaint() не забудьте вызвать**

**// реализацию базового класса.**

  base.OnPaint(e);

**// Получение объекта Graphics из поступившего на вход**

**// PaintEventArgs.**

  Graphics g = e.Graphics;

**// Визуализация текстового сообщения с заданными**

**// цветом и шрифтом.**

  g.DrawString("Привет GDI + ", new Font("Times New Roman", 20), Brushes. Green, 0, 0);

 }

}

Но, хотя переопределение OnPaint() и допустимо, более типичным подходом является обработка события Paint с помощью связанного делегата PaintEventHandler (именно это делается по умолчанию в Visual Studio 2005 при обработке событий с помощью окна свойств). Данный делегат может указывать на любой метод, получающий в качестве первого параметра System.Object, а в качестве второго – PaintEventArgs. В предположении о том, что вы обработали событие Paint (с помощью инструментов режима проектирования Visual Studio 2005 или в программном коде вручную), вы снова можете извлечь объект Graphics из поступающего на вход PaintEventArgs. Вот соответствующим образом модифицированный программный код.

public partial class MainForm: Form {

 public MainForm() {

  InitializeComponent();

  CenterToScreen();

  this.Text = "Basic Paint Form";

**// В Visual Studio 2005 поместите этот программный код**

**// в InitializeComponent().**

  this.Paint += new PaintEventHandler(MainForm\_Paint);

 }

 private void MainForm\_Paint(object sender, PaintEventArgs e) {

  Graphics g = e.Graphics;

  g.DrawString("Привет GDI+", new Font("Times New Roman", 20), Brushes.Green, 0, 0);

 }

}

Независимо от того, как вы отвечаете на событие Paint, следует знать, что событие Раint генерируется всегда, когда окно становится "грязным". Вы, возможно, знаете, что окно считается "грязным", если переопределяется его размер, окно (или его часть) открывается из-под другого окна, или окно сначала минимизируется, а затем восстанавливается. Во все случаях, когда требуется перерисовка формы, платформа .NET гарантирует, что обработчик события Paint (или переопределенный метод OnPaint() будет вызван автоматичеcки.

**Обновление области клиента формы**

В ходе выполнения приложения GDI+ может возникнуть необходимость в явном вызове события Paint вместо ожидания того, что окно станет "естественно грязным". Например, вы создаете программу, которая позволяет пользователю выбрать подходящий рисунок из набора точечных изображений в пользовательском диалоговом окне. После закрытия диалогового окна нужна отобразить выбранный пользователем рисунок в области клиента формж. Очевидно, если ждать, когда окно станет "естественно грязным", пользователь не увидит изменений до того, как изменятся размеры окна или его часть откроется из-под другого окна. Чтобы вызвать перерисовку окна программно, просто вызовите наследуемый метод Invalidate().

public partial class MainForm: Form {

 …

 private void MainForm\_Paint(object sender, PaintEventArgs e) {

  Graphics g = e.Graphics;

**// Здесь выполняется визуализация изображения.**

 }

 private void GetNewBitmap() {

**// Отображение диалогового окна и получение нового образа.**

**// Перерисовка клиентской области.**

  Invalidate();

 }

}

Метод Invalidate() является перегруженным, чтобы вы могли указать прямоугольную область для перерисовки, а не перерисовывать все области клиента (что делается до умолчанию). Чтобы обновить только заданный прямоугольник слева вверху области клиента, вы можете использовать следующее.

**// Перерисовка прямоугольной части формы.**

private void UpdateUpperArea() {

 Rectangle myRect = new Rectangle(0, 0, 75, 150);

 Invalidate(myRect);

}

**Доступ к объекту Graphics вне обработчика Paint**

В некоторых редких случаях может понадобиться доступ к объекту Graphics вне контекста обработчика события Paint. Предположим, например, что нужно перерисовать небольшой круг с центром в точке (х, у), где был выполнен щелчок кнопки мыши. Чтобы получить действительный объект Graphics в рамках контекста обработчика событий MouseDown, можно, например, вызвать статический метод Graphics.FromHwnd(). Имея опыт использования Win32, вы можете знать, что HWND является структурой данных, представляющей окно Win32. В рамках платформы .NET наследуемое свойство Handle извлекает соответствующую структуру HWND, которую затем можно использовать в качестве параметра для Graphics. FromHwnd().

private void MainForm\_MouseDown(object sender, MouseEventArgs e) {

**// Получение объекта Graphics через Hwnd.**

 Graphics g = Graphics**.FromHwnd**(this.Handle);

**// Рисование круга 10\*10 по щелчку мыши.**

 g.FillEllipse(Brushes.Firebrick, e.X, e.Y, 10, 10);

**// Освобождение объектов Graphic, созданных напрямую.**

 g.Dispose();

}

Эта логика отображает круг за пределами обработчика OnPaint(), но очень важно понимать, что когда выполняется обновление формы, все такие круги стираются! Это разумно, поскольку соответствующая визуализация выполнялась в контексте события MouseDown. Значительно лучшим подходом Является создание в обработчике события MouseDown нового типа Point, который добавляется к некоторой внутренней коллекции (например, List‹T›), и только затем вызывается Invalidate(). Тогда обработчик события Раint может просто "пройти" по коллекции и перерисовать каждый Point.

public partial class MainForm: Form {

***//*Используется для хранения всех Point.**

 private List‹Point› myPts = new List‹Point›();

 publiс MainForm() {

  …

  this.MouseDown += new MouseEventHandler(MainForm\_MouseDown);

 }

 private void MainForm\_MouseDown(object sender, MouseEventArgs e) {

**// Добавление в коллекцию.**

  myPts.Add(new Point(e.X, e.Y));

  Invalidate();

 }

 private void MainForm\_Paint(object sender, PaintEventArgs e) {

  Graphics g = e.Graphics;

  g.DrawString("Привет GDI+", new Font("Times New Roman", 20), new SolidBrush(Color.Black), 0, 0);

  foreach(Point p in myPts) g.FillEllipse(Brushes.Firebrick, p.X, p.Y, 10, 10);

 }

}

При таком подходе уже отображенные круги будут оставаться на месте, поскольку графическая визуализация обрабатывается в рамках события Paint. На рис. 20.1 показано окно тестового запуска этого приложения.



Рис 20.1. Простое графическое приложение

**Исходный код**. Проект BasiсPaintForm размещен в подкаталоге, соответствующем главе 20.

**Освобождение объекта Graphics**

Если вы внимательно читали несколько последних страниц, то могли заметить, что в *некоторых*примерах программного кода непосредственно вызывается метод Dispose() объекта Graphics, тогда как в других примерах этого не делается. Поскольку тип Graphics работает с самыми разными неуправляемыми ресурсами, имеет смысл освободить указанные ресурсы как можно быстрее с помощью Dispose() (не дожидаясь, когда это сделает сборщик мусора в процессе финализации). То же самое можно сказать о любом типе, поддерживающем интерфейс IDisposable. При работе с объектами Graphics нужно придерживаться следующих правил.

• Если объект Graphics был создан вами непосредственно, после окончания его использования его следует освободить.

• Если вы ссылаетесь на существующий объект Graphics, его освобождать не следует.

Для того чтобы это стало более понятным, рассмотрите следующий обработчик события Paint.

private void MainForm Paint(object sender, PaintEventArgs e) {

**// Загрузка локального файла**\*.**jpg.**

 image myImageFile = Image.FromFile("landscape.jpg");

**// Создание нового объекта Graphics на основе изображения.**

 Graphics imgGraphics = Graphics.FromImage(myImageFile);

**// Визуализация новых данных.**

 imgGraphics.FillEllipse(Brushes.DarkOrange, 50, 50, 150, 150);

**// Нанесение изображения на форму.**

 Graphics g = e.Graphics;

 g.DrawImage(myImageFile, new PointF(0.0F, 0.0F));

**// Освобождение созданного нами объекта Graphics.**

 imgGraphics.Dispose();

}

На данном этапе обсуждения не беспокойтесь о том, что некоторые элементы программной логики GDI+ могут быть для вас не вполне понятны. Однако обратите внимание на то, что здесь объект Graphics получается из файла \*.jpg, загружаемого (с помощью статического метода Graphics.FromImage()) из локального каталога приложения. Поскольку объект Graphics создается явно, после окончания использовании этого объекта лучше использовать Dispose(), чтобы освободить внутренние ресурсы и сделать их снова доступными для использования другими компонентами системы.

Однако, заметьте, Dispose() не вызывается явно для объекта, который был получен из поступающего на вход PaintEventArgs. Причина в том, что вы не создавали этот объект непосредственно и поэтому не можете гарантировать, что другие части программы не используют его. Очевидно, что при освобождении объекта, используемого в другом месте программы, могут возникать проблемы.

В связи с этим заметим, что если вы забудете вызвать метод Dispose() для объекта, реализующего IDisposable, внутренние ресурсы будут освобождены позже, при обработке объекта сборщиком мусора (см. главу 5). В этом смысле освобождение объекта imgGraphics, строго говоря, не является необходимым. Так что, хотя явное освобождение объектов GDI+, созданных вами непосредственно, делает программный код совершеннее, мы с вами, чтобы сделать примеры программного кода в этой главе более краткими, не будем освобождать каждый тип GDI+ вручную.

**Системы координат GDI+**

Нашей следующей задачей будет рассмотрение координатных систем GDI+. В GDI+ определяются три разные системы координат, которые используются средой выполнения, при определении места размещения и размеров содержимого визуализации. Во-первых, есть так называемые *мировые координаты*(или внешние координаты). Мировые координаты представляют абстракцию размеров данного типа GDI+, независимую от единиц измерения. Например, при прорисовке прямоугольника с указанием для размерности (0, 0, 100, 100), вы на самом деле указываете прямоугольник размером 100×100 "единиц". Как вы можете догадаться, по умолчанию "единица" – это пиксель, однако ей можно назначить и другую единицу измерения (дюйм, сантиметр и т.п.),

Далее, есть *страничные координаты*(координаты страницы). Страничные координаты представляют смещение в применении к оригинальным мировым координатам. Это удобно тогда, когда вы не хотите вычислять смещение в своем программном коде вручную (вы не обязаны это делать). Например, если у вас есть форма, которая должна оставаться в границах 100×100 пикселей, вы можете указать страничную координату (100\*100), чтобы визуализация выполнялась относительно точки (100\*100). Тогда в своем базовом коде вы сможете просто указать мировые координаты (избежав, тем самым, необходимости вручную учитывать смещение),

Наконец, есть *приборные координаты*(координаты устройства). Приборные координаты представляют результат применения страничных координат к оригинальным мировым координатам. Эта координатная система используется для определения того, где именно будет показан соответствующий тип GDI+. При программировании с помощью средств GDI+ программист обычно мыслит в терминах мировых координат, которые являются базой для определения размеров и места размещения типа GDI+. Для визуализации в мировых координатах не требуется никаких специальных программных ухищрений – нужно просто передать значения измерений текущей операции визуализации,

void MainForm\_Paint(object sender, PaintEventArgs s) {

**// Визуализация прямоугольника в мировых координатах.**

 i.Graphics g *=*е.Graphics;

 g.DrawRectangle(Pens.Black, 10, 10, 100, 100);

}

"За кулисами" ваши мировые координаты автоматически отображаются в координаты страницы, которые затем отображаются в приборные координаты. Во многих случаях вы вообще не будете использовать координаты страницы и приборные координаты непосредственно, если только не захотите применить определенные графические трансформации. Поскольку в предыдущем программном коде не используется никакой программной логики трансформаций, мировые, страничные и приборные координаты оказываются идентичными.

Если перед визуализацией своей программной логики GDI+ вы хотите применить какие-то преобразования, вы должны использовать подходящие члены типа Graphics (например, метод TranslateTransform()), чтобы перед тем, как выполнить визуализацию, указать "страничные координаты" в существующей системе мировых координат. В результате устанавливаются приборные координаты, которые будут использоваться при выводе типа GDI+ на соответствующее устройство.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

**// Указание смещения (10 \* 10) для страничных координат.**

 Graphics g = е.Graphics;

 g.TranslateTransform(10, 10);

 g.DrawRectangle(10, 10, 100, 100);

}

В данном случае при отображении прямоугольника его левый верхний угол фактически будет помещен в точку (20, 20), поскольку к мировой системе координат будет добавлено смещение в результате вызова TranslateTransform().

**Единица измерения, предлагаемая по умолчанию**

В GDI+ единицей измерения по умолчанию является пиксель. Начало координат размещается в левом верхнем углу с увеличением оси абсцисс вправо, а оси ординат – вниз (рис. 20.2).

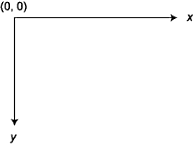


Рис. 20.2. Система координат GDI+, предлагаемая по умолчанию

Поэтому, если вы отобразите Rectangle с использованием пера толщиной в 5 пикселей и красного цвета, как показано ниже.

void MainForm\_Paint (object sender, PaintEventArgs e) {

**// Установка мировых координат с использованием единиц измерения,**

**// предлагаемых по умолчанию.**

 Graphics g = е.Graphics;

 g.DrawRectangle(newPen(Color.Red, 5), 10, 10, 100, 100);

}

вы должны увидеть квадрат, смещенный на 10 пикселей вниз и вправо относительно верхнего и левого края клиентской области формы, как показано на рис. 20.3.

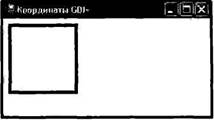


Рис. 20.3. Визуализация в пиксельных единицах

**Выбор альтернативной единицы измерения**

Если вы не хотите выполнять визуализацию изображений с использованием пиксельных единиц измерения, вы имеете возможность изменить эту принятую по умолчанию установку с помощью свойства PageUnit объекта Graphics. Свойству PageUnit можно присвоить любое значение из перечня GraphicsUnit.

public enum GraphicsUnit {

**// Мировые координаты.**

 World,

**// Пиксель для видеодисплея и 1/100 дюйма для принтера.**

 Display,

**// Пиксель.**

 Pixel,

**// Стандартная точка принтера (1/72 дюйма).**

 Point,

**// Дюйм.**

 Inch,

**// Стандартная единица документа (1/300 дюйма).**

 Document,

**// Миллиметр.**

 Millimeter

}

Чтобы проверить, как изменяется базовая единица измерения, модифицируйте имеющийся программный код так, как предлагается ниже.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

**// Отображение прямоугольника а дюймах, а не в пикселях…**

 Graphics g = e.Graphics;

 g.PageUnit = GraphicsUnit.Inch;

 g.DrawRectangle(new Pen(Color.Red, 5), 0, 0, 100, 100);

}

Вы должны увидеть совершенно другой прямоугольник, как показано на рис. 20.4.



Рис. 20.4. Визуализация в дюймах

Причина того, что здесь более 90% области клиента формы занято темным (красным) цветом, заключается в указании пера "шириной" в 5 дюймов! Сам прямоугольник теперь имеет размеры 100×100 дюймов, и тот маленький светлый прямоугольник, который вы видите на рисунке в правом нижнем углу, является левым верхним углом большого внутреннего прямоугольника.

**Изменение начала координат**

Напомним, что при использовании координат и единиц измерения, предлагаемых по умолчанию, точка (0, 0) находится в левом верхнем углу соответствующей области. Часто это и является именно тем, что требуется, но что делать, если вам нужно поменять точку, относительно которой выполняется визуализация? Предположим, например, что ваше приложение всегда должно (по какой-то причине) оставлять пустой полосу шириной в 100 пикселей вдоль границы области клиента формы. Тогда вы должны гарантировать, чтобы все операции GDI+ выполнялись в соответствующих пределах внутренней области,

Один из подходов, который можно при этом использовать, заключается в добавлении смещения вручную. Конечно, утомительно добавлять значения смещениях каждой операции визуализации. Значительно удобнее (и проще) было бы использовать свойство, которое, по сути, говорило бы следующее: "Хотя я даю указание отобразить прямоугольник с началом координат в точке (0, 0), вы должны использовать для начала координат точку (100, 100)". Это должно сильно "упростить вам жизнь", поскольку вы сможете указать параметры размещения без модификаций,

В рамках GDI+ вы можете указать точку начала координат, установив значение трансформации с помощью метода TranslateTransform() (класса Graphics), позволяющего указать страничные координаты, которые будут применяться к вашим оригинальным мировым координатам, например:

void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

**// Установка смещения (100, 100) для страничных координат.**

 g.TranslateTransform(100, 100);

**// Значениями мировых координат остаются (0, 0, 100, 100),**

**// но приборные координаты теперь равны (100, 100, 200, 200).**

 g.DrawRectangle(new Pen(Color.Red, 5), 0, 0, 100, 100);

}

Здесь вы устанавливаете для мировых координат значения (0, 0, 100, 100). Однако для страничных координат вы указали смещение (100, 100). Поэтому для приборных координат будут использоваться значений (100, 100, 200, 200). Таким образам, хотя вызов DrawRectangle() выглядит так, как будто вы размещаете прямоугольник в левом верхнем углу формы, результат будет выглядеть так, как показано на рис. 20.5.

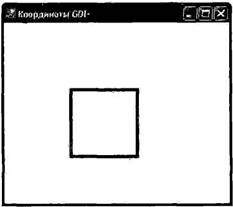


Рис. 20.5. Результат применения смещения страницы

Чтобы вы имели возможность поэкспериментировать с некоторыми способами изменения координатной системы GDI+, в файле с исходным текстом примеров этой книги (для его загрузки посетите раздел загрузки Web-узла Apress, размещенного по адресу [www.aprеss.com](http://www.xn--aprss-1we.com/)) есть пример приложения с именем CoorSystem. В этом приложении с помощью двух меню вы можете менять начало координат и единицу намерения (рис. 20.6).



Рис. 20.6. Изменение начала координат и единицы изменения

Теперь, когда вы имеете представление о роли базовых трансформаций для определения координат визуализации имеющегося GDI-типа на целевом устройстве, следующей задачей нашего обсуждения будет работа с цветовыми значениями.

**Исходный код.**Проект CoorSystem размещен в подкаталоге, соответствующем главе 20.

**Определение цветовых значений**

Многие методы визуализации, определенные классом Graphics, требуют от вас указания цвета, который должен использоваться в процессе рисования. Структура System.Drawing.Color представляет цветовую константу ARGB (от Alpha-Red-Green-Blue – альфа, красный, зеленый, синий). Функциональные возможности типа Color (цвет) представляются рядом статических доступных только для чтения свойств, возвращающих конкретный тип Color.

**// Один из множества встроенных цветов…**

Color с = Color.PapayaWhip;

Если стандартные цветные значения вам не подойдут, вы можете создать новый тип Color и указать для него значения A, R, G и В, используя метод FromArgb().

**// Указание ARGB вручную.**

Color myColor = Color.FromArgb(0, 255, 128, 64);

Используя метод FromName(), вы можете также сгенерировать тип Color по данному строковому значению. Строковый параметр должен при этом соответствовать одному из членов перечня KnownColor (который содержит значения для различных цветовых элементов Windows, например, таких как KnownColor.WindowFrame и KnownColor.WindowText).

**// Получение Color по известному имени.**

Color myColor = Color.FromName("Red")*;*

Независимо от метода получения типа Color, с этим типом можно взаимодействовать с помощью его членов.

• GetBrightness() – возвращает значение яркости типа Color на основании измерения HSB (Hue-Saturation-Brightness – оттенок, насыщенность, яркость).

• GetSaturation() – возвращает значение насыщенности типа Color на основании измерения HSB.

• GetHue() – возвращает значение оттенка типа Color на основании измерения HSB.

• IsSystemColor – индикатор того, что данный тип Color является зарегистрированным системным цветом.

• A, R, G, В – возвращают значения, присвоенные для альфа, красной, зеленой и синей составляющих типа Color.

**Класс ColorDialog**

Чтобы обеспечить конечному пользователю приложения возможность конфигурировать тип Color, пространство имен System.Windows.Forms предлагает встроенный класс диалогового окна с именем ColorDialog (рис. 20.7).

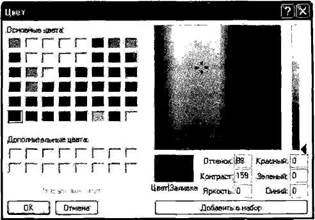


Рис. 20.7. Диалоговое окно настройки цветов Windows Forms

Работать с этим диалоговым окном очень просто. Для действительного экземпляра типа ColorDialog вызовите ShowDialog(), чтобы отобразить диалоговое окно модально. После закрытия диалогового окна пользователем вы сможете извлечь соответствующей объект Color, используя свойство ColorDialog.Color.

Предположим, что вы хотите с помощью ColorDialog предоставить пользователю возможность выбрать цвет фона для области клиента формы. Чтобы упростить ситуацию, мы будем отображать ColorDialog тогда, когда пользователь щелкнет в любом месте области клиента.

public partial class MainForm: Form {

 private = ColorDialog colorDlg;

 private Color currColor = Color.DimGray;

 public mainForm() {

  InitializeComponent();

  colorDlg = new ColorDialog();

  Text = "Для изменения цвета щелкните здесь";

  this.MouseDown += new MouseEventHandler(MainForm\_MouseDown);

 }

 private void MainForm\_MouseDown(object sender, MouseEventArgs e) {

  if (colorDlg.ShowDialog() ! = DialogResult.Cancel) {

   currColor = colorDlg.Color;

   this.BackColor = currColor;

   string strARGB = ColorDlg.Color.ToString();

   MessageBox.Show(strARGB, "Выбранный цвет ");

  }

 }

}

**Исходный код.**Проект ColorDIg размещен в подкаталоге, соответствующем главе 20.

**Манипулирование шрифтами**

Теперь давайте выясним, как можно программно манипулировать шрифтами. Тип System.Drawing.Font представляет шрифт, установленный на машине пользователя, Типы шрифта могут определяться с помощью любого числа перегруженных конструкторов. Вот вам несколько примеров.

**// Создание Font с заданными именем типа и размером.**

Font f = new Font("Times New Roman", 12);

**// Создание Font, с заданными именем типа, размером и начертанием.**

Font f2 = new Font("WingDings", 50, FontStyle.Bold | FontStyle.Underline);

При создании f2 здесь используются связанные с помощью операции OR значения из перечня FontStyle.

public enum FontStyle {

 Regular, Bold,

 Italic, Underline, Strikeout

}

После установки параметров объекта Font следующей вашей задачей должна быть передача этого объекта методу Graphics.DrawString() в виде параметра. Хотя метод DrawString() перегружен, каждая из его вариаций требует одну и ту же информацию: отображаемый текст, шрифт для отображения этого текста, кисть, с помощью которой выполняется визуализация, и место, в которое нужно текст поместить.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

**// Аргументы (String, Font, Brush, Point).**

 g.DrawString("Моя строка", new Font("WingDings", 25), Brushes.Black, new Point(0,0));

**// Аргументы (String, Font, Brush, int, int)**

 g.DrawString("Другая строка", new Font("Times New Roman", 16), Brushes.Red, 40, 40);

}

**Работа с семействами шрифтов**

Пространство имен System.Drawing определяет также тип FontFamily, предлагающий абстракцию для группы гарнитур, имеющих одинаковый базовый дизайн, но с определенными вариациями стиля. Семейство шрифтов, например, такое как Verdana, может включить в себя несколько шрифтов, отличающихся по стилю и размеру. Например, Verdana Bold (полужирный) 12 пунктов и Verdana Italic (курсив) 24 пункта являются разными шрифтами в рамках одного семейства шрифтов Verdana.

Конструктор типа FontFamily получает на вход строку с именем семейства шрифтов, которое вы пытаетесь представить. После создания "общего семейства" вы можете создать более специфичный объект Font.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g*=*e.Graphics;

**// Создание семейства шрифтов.**

 FontFamily myFamily = new FontFamilу("Verdana");

**// Передача семейства конструктору Font.**

 Font myFont = new Font(myFamily, 12);

 g.Drawstring("Привет!", myFont, "Brushes.Blue, 10*,*10);

}

Больший интерес представляет собой возможность сбора статистики в отношении данного семейства шрифтов. Скажем, вы создаете приложение текстового редактора и хотите определить среднюю ширину символа в конкретном объекте FontFamily. Или, например, вам нужна информация о надстрочных и подстрочных значениях для данного символа. Для получения такой информации тип FontFamily предлагает использовать специальные члены, описания которых приведены в табл. 20.5.

Таблица 20.5. Члены типа FontFamily

|  |  |
| --- | --- |
| **Член** | **Описание** |
| GetCellAscent() | Возвращает метрику надстрочного элемента для членов данного семейства |
| SetCellDescent() | Возвращает метрику подстрочного элемента для членов данного семейства |
| GetLineSpacing() | Возвращает расстояние между двумя последовательными строками текста для данного FontFamily с указанным FontStyle |
| GetName() | Возвращает имя данного FontFamily на указанном языке |
| IsStyleAvailable() | Индикатор доступности указанного FontStyle |

Для примера рассмотрите следующий обработчик события Paint, выводящий на печать ряд 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 у = 0;

 int fontHeight = myFont.Height;

**// Отображение единицы измерения для членов FontFamily.**

 this.Text = "Единица измерения: GraphicsUnit." + myFont.Unit;

 g.DrawString("Семейство Verdana.", myFont, Brushes.Blue, 10, y);

 у += 20;

**// Характеристики связей нашего семейства.**…

 g.DrawString("Надстрочные для Verdana Bold: " + myFamily.GetCellAscent(FontStyle.Bold), myFont, Brushes.Black, 10, у + fontHeight);

 у += 20;

 g.DrawString("Подстрочные для Verdana Bold: " + myFamily.GetCellDescent(FontStyle.Bold), myFont, Brushes.Black, 10, у + fontHeight);

 у += 20;

 g.DrawString("Интерлиньяж для Verdana Bold: " + myFamily.GetLineSpacing(FontStyle.Bold), myFont, Brushes.Black, 10, у + fontHeight);

 у += 20;

 g.DrawString("Высота для Verdana Bold: " + myFamily.GetEmHeight(FontStyle.Bold), myFont, Brushes.Black, 10, у + fontHeight);

 у += 20;

}

На рис. 20.8 показан результат.



Рис. 20.8. Сбор статистики для семейства шрифтов Verdana

Заметьте, что указанные члены типа Font Family возвращают значения с использованием в качестве единицы измерения GraphicsUnit.Point (а не Pixel), что соответствует 1/72 дюйма. Вы можете преобразовать эти значения в те единицы, которые вам подходят лучше всего.

**Исходный код.**Проект FontFamilyApp размещен в подкаталоге, соответствующем главе 20.

**Работа с гарнитурами и размерами шрифтов**

Давайте теперь построим более сложное приложение. Позволяющее пользователю манипулировать объектом Font, поддерживаемым формой. Это приложение предоставит пользователю возможность указать гарнитуру шрифта, используя встроенный набор гарнитур, доступный путем выбора Сервис→Гарнитура из меню. Пользователю также будет позволено косвенно управлять размером объекта Font с помощью объекта Timer Windows Forms. Если пользователь активизирует Timer, выбрав из меню Сервис→Рост?, то размер объекта Font начнет увеличиваться (до максимального верхнего предела) через регулярные интервалы времени. При этом отображаемый текст будет постепенно увеличиваться, что обеспечит анимационный эффект "живого текста". Наконец, третий элемент меню Сервис будет называтъся Список шрифтов и показывать список всех шрифтов, установленных на машине конечного пользователя. На рис. 20.9 демонстрируется логика меню, о котором идет речь.

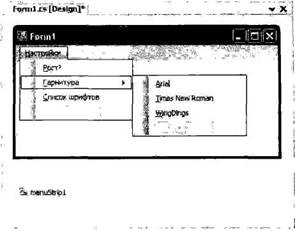


Рис. 20.9. Меню проекта FontApp

Чтобы начать реализацию приложения, добавьте в форму член Timer (с именем swellTimer), строку (strFontFace) для представления текущего названия гарнитуры шрифта и целое число (swellValue) для представления величины корректировки для размера шрифта. В окне проектирования формы сконфигурируйте Timer так, чтобы он генерировал событие Tick каждые 100 миллисекунд.

public partial class MainForm: Form {

 private Timer swellTimer = new Timer();

 private int swellValue;

 private string strFontFace = "WingDings";

 public MainForm() {

  InitializeComponent();

  BackColor = Color.Honeydew;

  CenterToScreen();

**// Конфигурация таймера.**

  swellTimer.Enabled = true;

  swellTimer.Interval = 100;

  swellTimer.Tick += new EventHandler(swellTimerTick);

 }

}

В обработчике события Tick увеличьте значение члена swellValue на 5. Напомним, что целое число swellValue будет добавляться к текущему размеру шрифта, чтобы обеспечивался простой эффект анимации (предполагается, что swellValue будет ограничено сверху максимальным значением 50). Чтобы не допустить мерцания, которое может происходить при перерисовке всей области клиента, при вызове Invalidate() будет обновляться только верхняя прямоугольная область формы.

private void swellTimer Tick(object sender, EventArgs e) {

**// Увеличение текущего значения swellValue на 5.**

 swellValue += 5;

**// Если значение больше или равно 50, сбросить его в ноль.**

 if (swellValue ›= 50) swellValue = 0;

**// Обновление минимальной области для уменьшения мерцания.**

 Invalidate(new Rectangle(0, 0, ClientRectangle.Width, 100));

}

Теперь, когда с каждым циклом Timer обновляются верхние 100 пикселей области клиента, нужно найти что-нибудь подходящее для визуализации. В обработчике Paint формы создайте объект Font на основе выбранной пользователем гарнитуры шрифта (она выбирается с помощью соответствующего пункта меню) и текущего значения swellValue (оно задается таймером Timer), Настроив объект Font, поместите сообщение в центр соответствующего прямоугольника.

private void MainForm Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

**// Размер шрифта должен находиться в диапазоне от 12 до 62,**

**// в зависимости от swellValue.**

 Font theFont = new Font(strFontFace, 12 + swellValue);

 string message = "Привет GDI+";

**// Вывод сообщения в центре прямоугольника.**

 float windowCenter = this.DisplayRectangle.Width/2;

 SizeF stringSize = g.Measure.String(message, theFont);

 float startPos = windowCenter – (stringSize.Width/2);

 g.Drawstring(message, theFont, new SolidBrush(Color.Blue), startPos, 10);

}

Легко догадаться, что при выборе пользователем конкретной гарнитуры шрифта обработчик Clicked для соответствующего варианта выбора из меню должен обновить строковую переменную fontFace и перерисовать область клиента, например:

private void arialToolStripMenuItem\_Click(object sender, EventArgs e) {

 strFontFace = "Arial";

 Invalidate();

}

Обработчик Click для пункта меню Рост? будет использоваться для запуска и остановки процесса увеличения текста (т.е. для разрешения и отключения анимаций). Здесь используйте свойство Enabled объекта Timer так, как показано ниже.

private void swellToolStripMenuItem\_Click(object sender, EventArgs e) {

 swellTimer.Enabled = !swellTimer.Enabled;

}

**Список установленных шрифтов**

Давайте расширим программу так, чтобы она отображала множество установленных на машине шрифтов с помощью типов из пространства имен System.Drawing.Text. Это пространство имен содержит набор типов, которые можно использовать для получения списка шрифтов, установленных на целевой машине, и для работы с ними. Дня наших целей достаточно рассмотреть только класс InstalledFontCollection.

Когда пользователь выбирает из меню Сервис→Список шрифтов, соответствующий обработчик Clicked создает экземпляр класса InstalledFormCollection. Этот класс содержит массив FontFamily, представляющий набор всех шрифтов, установленных на целевой машине, и этот массив можно получить, используя свойство InstalledFontCollection.Families. С помощью свойства FontFamily.Name вы можете извлечь название гарнитуры шрифта (например, Times New Roman, Arial и т.п.).

Добавьте в форму приватный член-строку с именем installedFonts для хранения названия гарнитур. Программная логика обработки пункта меню Список Шрифтов создает экземпляр типа InstalledFontCollection, читает имя каждого элемента и добавляет новую гарнитуру в приватный член installedFonts.

public partial class MainForm: Form {

**// Содержит список шрифтов.**

 private string installedFonts;

**// Обработчик меню для получения списка шрифтов.**

 private void mnuConfigShowFonts\_Clicked(object sender, EventArgs e) {

  InstalledFontCollection fonts = new InstalledFontCollection();

  for (int i = 0; i ‹ fonts.Families.Length; i++) installedFonts += fonts.Families[i].Name + " ";

**// На этот раз нужно обновить всю область клиента,**

**// поскольку обновляется строка installedFonts в нижней части**

**// области клиента.**

  Invalidate();

 }

}

Заключительной нашей задачей будет отображение строки installedFonts в области клиента, расположенной сразу же под той частью экрана, которая исполь-зуетcя для движущегося текста.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

 Font theFont = new Font(strFontFace, 12 + swellValue);

 string message = "Привет GDI+";

**// Отображение сообщения в центре окна.**

 float windowCenter = this.DisplayRectangle.Width/2;

 SizeF.stringSize = e.Graphics.MeasureString(message, theFont);

 float startPos = windowCenter – (stringSize.Width/2);

 g.DrawString(message, theFont, Brushes.Blue, startPos, 10);

**// Показ списка установленных шрифтов в прямоугольнике**

**// под движущимся текстом.**

 Rectangle myRect = new Rectangle(0, 100, ClientRectangle.Width, ClientRectangle.Height);

**// Закрашивание данной области формы черным цветом.**

 g.FillRectangle(new SolidBrush(Color.Black), myRect);

 g.DrawString(installedFonts, new Font("Arial", 12), Brushes.White, myRect);

}

Напомним, что размеры "грязного прямоугольника" проецировались в верхние 100 пикселей области клиента. Поскольку обработчик Tick обновляет только часть формы, остальная ее часть при посылке события Tick не перерисовывается (чтобы оптимизировать задачу визуализации в области клиента).

В качестве завершающего штриха давайте обработаем событие Resize формы, чтобы гарантировать соответствующую перерисовку в нижней части прямоугольника клиента в том случае, когда пользователь изменяет размеры формы.

private void Main.Form\_Resize(object sender, System.EventArgs e) {

 Rectangle myRect = new Rectangle(0, 100, ClientRectangle.Width, ClientRectangle.Height);

 Invalidate(myRect);

}

На рис. 20.10 показан результат (с текстом, представленным шрифтом WingDings!).



Рис. 20.10. Приложение SwellingFontApp в действии

**Исходный код.**Проект SwellingFontApp размещен в подкаталоге, соответствующем главе 20.

**Класс FontDialog**

Как вы можете догадываться, существует и класс диалогового окна для настройки шрифтов (FontDialog). Вид этого окна показан на рис. 20.11.

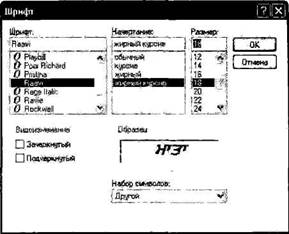


Рис. 20.11. Диалоговое окно *Шрифт*Windows Forms

Подобно типу ColorDialog, рассмотренному в этой главе выше, для работы с FontDialog нужно просто вызвать метод ShowDialog(). Используя свойства Font, можно извлечь текущие характеристики шрифта для использования в приложении. Для примера рассмотрите следующую форму, имитирующую логику предыдущего проекта ColorDlg. При щелчке пользователя в любом месте окна формы отображается диалоговое окно Шрифт и выводится информация о текущем выборе.

public partial class MainForm: Form {

 private FontDialog fontDlg = new FontDialog();

 private Font currFont = new Font("Times New Roman", 12);

 public MainForm() {

  InitializeComponent(); CenterToScreen();

 }

 private void MainForm\_MouseDown(object sender, MouseEventArgs e) {

  if (fontDlg.ShowDialog() != DialogResult.Cancel) {

   currFont = fontDlg.Font;

   this.Text = string.Format("Selected Font: {0}", currFont); Invalidate();

  }

 }

 private void MainForm\_Paint(object sender, PaintEventArgs e) {

  Graphics g = e.Graphics;

  g.DrawString("Проверка…", currFont, Brushes.Black, 0, 0);

 }

}

**Исходный код.**Проект FontDlgForm размещен в подкаталоге, соответствующем главе 20.

**Обзор пространства имен System.Drawing.Drawing2D**

Теперь, когда мы обсудили возможности использования типа Font, следующей нашей задачей будет рассмотрение объектов Pen и Brush, предназначенных для визуализации геометрических шаблонов. Вы, конечно, можете ограничиться использованием только вспомогательных типов Brushes и Pens для получения уже сконфигурированных типов со сплошным цветом, но вы должны знать о том, что в пространстве имен System.Drawing.Drawing2D есть очень много и более "экзотических" перьев и кистей,

Это дополнительное пространство имен GDI+ предлагает ряд классов, позволяющих изменить форму пера (треугольник, ромб и т.д.), указать текстуру кисти и работать с векторной графикой. Некоторые базовые типы, о которых вам следует знать (сгруппированные по функциональным возможностям), описаны в табл. 20.6.

**Таблица 20.6.**Классы System.Drawing.Drawing2D

|  |  |
| --- | --- |
| **Классы** | **Описание** |
| AdjustableArrowCap CustomLineCap | Используются для изменения формы концов линий для перьев, Данные типы задают, соответственно, регулируемую стрелку и пользовательскую форму конца линии |
| Blend ColorBlend | Позволяют определить шаблон смешивания (и цвет) для использования с LinearGradientBrush |
| GraphicsPath GraphicsPathIterator PathData | Объект GraphicsPath представляет серию линий и кривых. Этот класс позволяет добавлять в траектории геометрические шаблоны практически любого вида (дуги, прямоугольники, линии, строки, многоугольники и т.д.). PathData содержит графические данные, формирующие траекторию |
| HatchBrush LinearGradientBrush PathGradientBrush | Экзотические типы кистей |

Также следует знать о том. что пространство имен System.Drawing.Drawing2D определяет набор перечней (DashStyle, FillMode, HatchStyle, LineCap и т.д.), которые используются вместе с указанными в таблице базовыми типами.

**Работа с типами Pen**

Типы Pen GDI+ используются для построения линий, соединяющих конечные точки. Сам по себе тип Pen не слишком полезен. Для выполнения визуализации геометрической формы на поверхности производного от Control типа действительный тип Pen следует направить подходящему методу визуализации, определенному классом Graphics. Вообще говоря, с объектами Pen обычно используются методы DrawXXX(), позволяющие отобразить некоторый набор линий на соответствующей графической поверхности.

Тип Pen определяет небольшой набор инструкторов, позволяющих задать начальный цвет и ширину пера. По большей части функциональные возможности Pen задаются свойствами, поддерживаемыми данным типом. Описания некоторых из этих свойств предлагаются в табл. 20.7.

**Таблица 20.7.**Свойства Pen

|  |  |
| --- | --- |
| **Свойства** | **Описание** |
| Brush | Определяет тип Brush для использования с данным типом Pen |
| Color | Определяет тип Color для использования с данным типом Pen |
| CustomStartCap CustomEndCap | Читает или устанавливает параметры пользовательского стиля концов линий, создаваемых с помощью данного типа Pen. *Стиль концов линий* – это просто термин, используемый для обозначения того, как должен выглядеть начальный и заключительный 'штрих" данного пера. Эти свойства позволяют строить пользовательские стили начала и конца линий для типов Pen |
| DashCap | Читает или устанавливает параметры стиля концов линий, используемого для прерывистых линий, создаваемых с помощью данного типа Pen |
| DashPattern | Читает или устанавливает массив пользовательской маски для рисования прерывистых линий. Соответствующие "тире" складываются из сегментов линий |
| DashStyle | Читает или устанавливает параметры стиля, используемого для прерывистых линий, создаваемых с помощью данного типа Pen |
| StartCap EndCap | Читает или устанавливает встроенный стиль концов линий, создаваемых с помощью данного типа Pen. Стиль концов линий Pen устанавливается в соответствии с перечнем LineCap, определенным в пространстве имен System.Drawing.Drawing2D |
| Width | Читает или устанавливает ширину данного Pen |
| DashOffset | Читает или устанавливает расстояние от начала линии до начала шаблона прерывистой линии |

Помните о том, что вдобавок к типу Pen в GDI+ предлагается коллекция Pens. Используя ряд статических свойств, вы можете извлечь из этой коллекции объект Pen (или нужный цвет) "на лету", не создавая пользовательский тип Pen вручную. Однако следует знать*,*что возвращаемый: при этом тип Pen всегда имеет ширину, равную 1.

Если вам понадобятся какие-то экзотические перья, вы должны создать тип Pen вручную. Предоставив вам эту информацию, я теперь могу перейти к построению примеров геометрических изображений с помощью простых типов Pen. Предположим, что у нас есть главный объект Form, способный отвечать на запросы визуализации. Реализация ответа выглядит так.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g*=*e.Graphics;

**// Создание большого пера синего цвета.**

 Pen bluePen = new Pen(Color.Blue, 20);

**// Получение готового пера из типа Pens.**

 Pen pen2 = Pens.Firebrick;

**// Визуализация некоторых шаблонов.**

 g.DrawEllipse(bluePen, 10, 10, 100, 100);

 g.DrawLine(pen2, 10, 130, 110, 130);

 g.DrawPie(Pens.Black, 150, 10, 120, 150, 90, 80);

**// Рисование пурпурного полигона с пунктирной границей.…**

 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 Point(190, 190), new Point(50, 330), new Point(20, 180) });

**//… и прямоугольника, содержащего текст.…**

 Rectangle r = new Rectangle (150, 10, 130, 60);

 g.DrawRectangle(Pens.Blue, r);

 g.DrawString("Эй, вы, там, наверху!… Я вам привет передаю.", new Font("Arial", 11), Brushes.Black, r);

}

Заметьте, что тип Pen, применяемый для отображения многоугольника, использует перечень DashStyle (определенный в System.Drawing.Drawing2D).

public enum DashStyle {

 Solid, Dash, Dot,

 DashDot, DashDotDot, Custom

}

В дополнение к уже имеющимся элементам DashStyle вы можете определить пользовательские шаблоны, используя для этого свойство DashPattern типа Pen.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

 …

**// Рисование прерывистой линии вдоль границы формы**

**// по пользовательскому шаблону.**

 Pen customDashPen = new Pen(Color.BlueViolet, 10);

 float[] myDashes = {5.0f, 2.0f, 1.0f, 3.0f};

 customDashPen.DashPattern = myDashes;

 g.DrawRectangle(customDashPen, ClientRectangle);

}

На рис. 20.12 показан вывод этого варианта обработчика событий Paint.

**Исходный код**. Проект CustomPenApp размещен в подкаталоге, соответствующем главе 20.

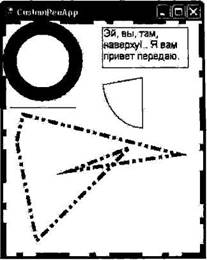


Рис. 20.12. Работа с типами Pen

**Концы линий**

Если рассмотреть вывод предыдущего примера, вы должны заметить, что начало и конец каждой линии там оформлен вполне стандартно – линия "срезается" под углом 90° к ее направлению. Но, используя перечень LineCap, вы имеете возможность создавать объекты Pen, демонстрирующие иное поведение.

public enum LineCap {

 Flat, Square, Round,

 Triangle, NoAnchor,

 SquareAnchor, RoundAnchor,

 DiamondAnchor, ArrowAnchor,

 AnchorMask, Custom

}

Следующее приложение отображает набор линий, по очереди используя каждый из стилей LineCap. Конечный результат показан на рис. 20.13.

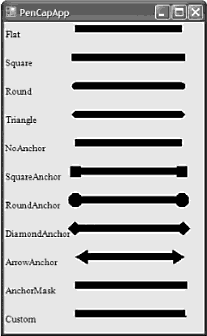


Рис. 20.13. Работа с концами линий

В соответствующем программном коде просто используется цикл по всем членам перечня LineCap с выводам имени соответствующего элемента (например, ArrowAnchor) и отображением линии с соответствующим оформлением ее концов.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

 Pen thePen = new Pen(Color.Black, 10);

 int yOffSet = 10;

**// Получение всех членов перечня LineCap.**

 Array obj = Enum.GetValues(typeof(LineCap));

**// Рисование линии для данного члена LineCap.**

 for (int х = 0; х ‹ obj. Length; x++) {

**// Получение следующего стиля конца линии и настройка пера.**

  LineCap temp = (LineCap)obj.GetValue(x);

  thePen.StartCap = temp;

  thePen.EndCap = temp;

**// Вывод имени из перечня LineCap.**

  g.Drawstring(temp.ToString(), new Font("Times New Roman", 10), new SolidBrush(Color.Black), 0, yOffSet);

**// Рисование линии с соответствующим стилем концов.**

  g.DrawLine(thePen, 100, yOffSet, Width – 50, yOffSet);

  yOffSet += 40;

 }

}

**Исходный код.**Проект PenCapApp размещен в подкаталоге, соответствующем главе 20.

**Работа с типами Brush**

Типы, производные от System.Drawing.Brush, используются для заполнения имеющегося региона заданным цветом, узором или изображением. Сам класс Brush является абстрактным типом, поэтому он не позволяет создать соответствующий экземпляр непосредственно. Однако Brush может играть роль базового класса для родственных ему типов кисти (например, SolidBrush, HatchBrush, LinearGradientBrush и т.д.). Кроме относящихся к Brush типов, пространство имей System.Drawing определяет также два вспомогательных класса, возвращающие кисти, уже сконфигурированные с помощью ряда статических свойств: это классы Brushes и SystemBrushes. Так или иначе, получив кисть, вы получаете возможность вызвать любой из методов FillXXX() типа Graphics.

Интересно то, что на основе кисти вы можете создать пользовательский тип Pen. Подобным образом вы можете создать себе любую подходящую кисть (кисть, которая "рисует" точечное изображение) и выполнять визуализацию заданных геометрических шаблонов с помощью сконфигурированного объекта Pen. Для примера рассмотрите следующий вариант программы, в котором используются различные кисти.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

**// Создание SolidBrush синего цвета*.***

 SolidBrush blueBrush = new SolidBrush(Color.Blue);

**// Получение готовой кисти из типа Brushes.**

 SolidBrush pen2 = (SolidBrush)Brushes.Firebrick;

**// Визуализация некоторых шаблонов.**

 g.FillEllipse(blueBrush, 10, 10, 100, 100);

 g.FillPie(Brushes.Black, 150, 10, 120, 150, 90, 80);

**// Рисование пурпурного полигона…**

 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 прямоугольника, содержащего текст**…

 Rectangle r = new Rectangle(150, 10, 130, 60);

 g.FillRectangle(Brushes.Blue, r);

 g.DrawString("Эй, вы, там, наверху!… Я вам привет передаю.", new Font("Arial", 11), Brushes.White, r);

}

Надеюсь, вы согласитесь, что это приложение почти идентично созданной выше программе CustomPenApp, но использует методы FillXXX() и типы SolidBrush вместо перьев и соответствующим им методов DrawXXX(). На рис. 20.14 показан соответствующий вывод.

**Исходный код**. Проект SolidBrushApp размещен в подкаталоге, соответствующем главе 20.



Рис. 20.14. Работа с типами Brush

**Работа с HatchBrush**

В пространстве имен System.Drawing.Drawing2D определен производный от Brush тип с именем HatchBrush. Этот тип позволяет закрасить регион, используя один из (очень большого) набора встроенных видов узоров, представленных перечнем HatchStyle. Вот часть соответствующего списка имен.

public enum HatchStyle {

 Horizontal, Vertical, ForwardDiagonal,

 BackwardDiagonal, Cross, DiagonalCross,

 LightUpwardDiagonal, DarkDownwardDiagonal,

 DarkUpwardDiagonal, LightVertical,

 NarrowHorizontal, DashedDownwardDiagonal,

 SmallConfetti, LargeConfetti, ZigZag,

 Wave, DiagonalВrick, Divot, DottedGrid, Sphere,

 OutlinedDiamond, SolidDiamond,

 …

}

При конструировании HatchBrush вы должны указать цвет для переднего плана и цвет для фона, которые будут использоваться при выполнении операции закрашивания. Для примера давайте немного подкорректируем программную логику из приведенного выше примера PenCapApp.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

 int yOffSet = 10;

**// Получение всех членов перечня HatchStyle.**

 Array obj = Enum.GetValues(typeof(HatchStyle));

**// Отображение овалов для первых 5 значений из HatchStyle.**

 for (int x = 0; x ‹ 5; x++) {

**// Конфигурация кисти.**

  HatchStyle temp = (HatchStyle)obj.GetValue(x);

  HatchBrush theBrush = new HatchBrush(temp, Color.White, Color.Black);

**// Вывод имени из перечня HatchStyle.**

  g.DrawString(temp.ToString(), new Font (''Times New Roman", 10), Brushes.Black, 0, yOffSet);

**// Закраска объекта подходящей кистью.**

  g.FillEllipse(theBrush, 150, yOffSet, 200, 25);

  yOffSet += 40;

 }

}

В окне вывода будут показаны заполненные овалы для первых пяти значений видов штриховки (рис. 20.15).

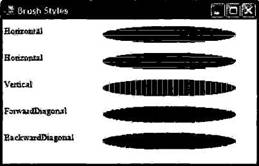


Рис. 20.15. Некоторые стили штриховки

**Исходный код.**Проект BrushStyles размещен в подкаталоге, соответствующем главе 20.

**Работа с TextureBrush**

Тип TextureBrush позволяет связать с кистью точечное изображение, чтобы затем использовать ее в операциях закрашивания. Чуть позже будет подробно обсуждаться класс image (изображение) GDI+. Типу TextureBrush предоставляется ссылка на image, используемая этим типом в течение всего цикла его существования. Само изображение обычно хранится в некотором локальном файле (\*.bmp. \*.gif, \*.jpg) или же встроено в компоновочный блок .NET.

Давайте построим пример приложения, использующего тип TextureBrush. Одна кисть будет использоваться дня закраски области клиента изображением из файла clouds.bmp, в то время как другая кисть будет выводить текст с помощью изображения, находящегося в файле Soap bubbles.bmp. Соответствующий вывод показан на рис. 20.16.

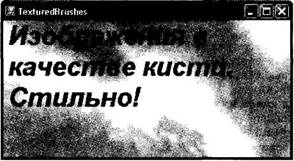


Рис. 20.16. Точечные рисунки в качестве кисти

Ваш производный от Form класс должен поддерживать два члена типа Brush, которым присваивается новый объект TextureBrush в конструкторе. Обратите внимание на то, что конструктору типа TextureBrush требуется предоставить на вход тип, производный от Image.

public partial class MainForm: Form {

**// Данные для кисти с изображением.**

 private Brush texturedTextBrush; private Brush texturedBGroundBrush;

 public MainForm() {

  …

**// Загрузка изображения для кисти фона.**

  Image bGroundBrushImage = new Bitmap("Clouds.bmp");

  texturedBGroundBrush = new TextureBrush(bGroundBrushImage);

**// Загрузка изображения для кисти текста.**

  Image textBrushImage = new Bitmap("Soap Bubbles.bmp");

  texturedTextBrush = new TextureBrush(textBrushImage);

 }

}

**Замечание.**Файлы \*.bmp, которые используются в этом примере, должны находиться в той же папке, где находится само приложение (или должны быть "жестко" указаны пути, по которым эти изображения можно найти). Соответствующая проблема будет обсуждаться в этой главе чуть позже.

Теперь, когда у вас есть два типа TextureBrush, способные выполнить визуализацию, создать обработчик события Paint очень просто.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

 Rectangle r = ClientRectangle;

**// Рисование облаков в области клиента.**

 g.FillRectangle(texturedBGroundBrush, r);

**// Отображение текста кистью с текстурой.**

 g.DrawString("Изображения в качестве кисти. Стильно!", new Font("Arial", 30, FontStyle.Bold | FontStyle.Italic), texturedTextBrush, r);

}

Исходный код. Проект TexturedBrushes размещен в подкаталоге, соответствующем главе 20.

**Работа с LinearGradientBrush**

Последним из рассматриваемых в этом разделе типов будет тип LinearGradientBrush, который можно использовать тогда, когда нужно смешать два цвета в градиентной закраске. Работать с этим типом так же просто, как и с остальными типами кисти. Важным моментом здесь Является то, что при создании LinearGradientBrush нужно указать пару типов Color и значение для направления смешивания из перечня LinearGradientMode.

public enum LinearGradientMode {

 Horizontal, Vertical,

 ForwardDiagonal, BaсkwardDiagonal

}

Чтобы проверить эти значения, с помощью LinearGradientBrush отобразим серию прямоугольников.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

 Rectangle r = new Rectangle(10, 10, 100, 100);

**// Градиентная кисть.**

 LinearGradientBrush theBrush = null;

 int yOffSet = 10;

**// Получение членов перечня LinearGradientMode**.

 Array obj = Enum.GetValues(typeof(LinearGradientMode));

**// Отображение прямоугольников для членов LinearGradientMode.**

 for(int x = 0; x ‹ obj.Length; x++) {

***//*Конфигурация кисти.**

  LinearGradientMode temp = (LinearGradientMode)obj.SetValue(x);

  theBrush = new LinearGradientBrush(r, Color.GreenYellow, Color.Blue, temp);

**// Вывод имени из перечня LinearGradientMode.**

  g.DrawString(temp.ToString(), new Font("Times New Roman", 10), new SolidBrush(Color.Black), 0, yOffSet);

**// Закраска прямоугольника подходящей кистью.**

  g.FillRectangle(theBrush, 150, yOffSet, 200, 50);

  yOffSet += 80;

 }

}

На рис. 20.17 показан результат.

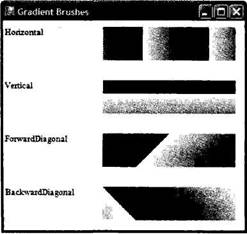


Рис. 20.17. Градиентная кисть за работой

**Исходный код.**Проект GradientBrushes размещен в подкаталоге, соответствующем главе 20.

**Визуализация изображений**

К этому моменту вы знаете, как работать с тремя из четырех главных типов GDI+: шрифтами, перьями и кистями. Заключительным типом, который мы с вами рассмотрим в этой главе, будет класс Image (изображение) и связанные с ним подтипы. Абстрактный тип System.Drawing.Image определяет ряд методов и свойств, хранящих различную информацию о том изображении, которое этот тип представляет. Например, для представления размеров изображения класс Image предлагает свойства Width, Height и Size. Другие свойства позволяют получить доступ к палитре изображения. Описания базовых членов класса Image приведены в табл. 20.8.

Таблица 20.8. Члены типа Image

|  |  |
| --- | --- |
| **Члены** | **Описание** |
| FromFile() | Статический метод, создающий объект Image из указанного файла |
| FromStream() | Статический метод, создающий объект Image из указанного потока данных |
| Height Width Size HorizontalResolution VerticalResolution | Свойства, возвращающие информацию о размерах данного объекта Image |
| Palette | Свойство, возвращающее тип данных ColorPalette, который представляет палитру, используемую для данного объекта Image |
| GetBounds | Метод, возвращающий объект Rectangle, который представляет текущие размеры данного объекта Image |
| Save() | Метод, сохраняющий в файл данные, содержащиеся в производном от Image типе |

Поскольку экземпляр абстрактного класса Image нельзя создать непосредственно, обычно непосредственно создается экземпляр типа Bitmap. Предположим, что у нас есть некоторый класс Form, отображающий три точечных рисунка в области клиента. Указав для каждого из типов Bitmap подходящий файл изображения, просто отобразите их в обработчике события Paint, используя метод Graphics.DrawImage().

public partial class MainForm: Form {

 private Bitmap[] myImages = new Bitmap[3];

 public MainForm() {

**// Загрузка локальных изображений.**

  myImages[0] = new Bitmap("imageA.bmp");

  myImages[1] = new Вitmap("imageB.bmp");

  myImages[2] = new Bitmap("imageC.bmp");

  CenterToScreen();

  InitializeComponent();

 }

 private void MainForm\_Paint(object sender, PaintEventArgs e) {

  Graphics g = e.Qraphics;

**// Визуализация изображений.**

  int yOffSet = 20;

  foreach (Bitmap b in myImages) {

   g.DrawImage(b, 10, yOffSet, 90, 90);

   yOffSet += 100;

  }

 }

}

**Замечание**. Файлы \*.bmp, которые используются в этом примере, должны находиться в той же папке, где находится само приложение (или должны быть "жестко" указаны пути, по которым эти изображения можно найти). Соответствующая проблема будет обсуждаться в этой главе чуть позже.

На рис. 20.18 показан соответствующий вывод.



Рис. 20.18. Визуализация изображений

Наконец, необходимо отметить, что, несмотря на имя Bitmap, этот класс может содержать изображения, сохраненные в любом из целого ряда форматов (\*.tif, \*.gif, \*.bmp и т.д.).

**Исходный код**. Проект BasicImages размещен в подкаталоге, соответствующем главе 20.

**Попадание в заданную область и операции перетаскивания для PictureBox**

Вы, конечно, можете отображать изображения Bitmap на поверхности любого производного класса Control непосредственно, но вы быстро обнаружите, что гораздо более широкие возможности и лучший контроль дает размещение изображения в рамках типа PictureBox. Например, поскольку тип PictureBox является производным от Control, он наследует соответствующие функциональные возможности, например, способность обрабатывать различные события, поддерживать всплывающие подсказки или контекстные меню и т.д. Такого поведения можно добиться и при непосредственном использовании Bitmap, но при этом вам придется создавать соответствующий программный код самостоятельно.

Чтобы продемонстрировать пользу типа PictureBox, давайте создадим простую "игру", которая будет способна "распознать" присутствие указателя мыши на графическом изображении. При щелчке пользователя кнопкой мыши в границах изображения включается режим "перетаскивания", и пользователь может перемещать изображение по форме. Чтобы сделать ситуацию интереснее, давайте проконтролируем, где пользователь "отпустит" изображение. Если это произойдет в рамках некоторого прямоугольника визуализации GDI+, будет выполнено некоторое заданное действие (оно будет описано чуть позже). Вы, возможно, знаете, что процесс обнаружения событий мыши в заданной области называется *проверкой попадания в заданную область.*

Тип PictureBox наследует большинство своих функциональных возможностей от базового класса Control. Ряд членов Control был уже рассмотрен в предыдущей главе, и это позволяет нам сразу перейти к обсуждению вопроса назначения изображения члену PictureBox с помощью свойства Image (снова заметим, что файл happyDude.bmp должен находиться в каталоге приложения).

public partial class MainForm: Form {

**// Содержит изображение улыбающегося лица.**

 private PictureBox happyBox = new PictureBox();

 public MainForm() {

**// Конфигурация PictureBox.**

  happyBox.SizeMode = PictureBoxSizeMode.StretchImage;

  happyBox.Locaton = new System.Drawing.Point(64, 32);

  happyBox.Size = new System.Drawing.Size(50, 50);

  happyBox.Cursor = Cursors.Hand;

  happyBox.Image = new Bitmap("happyDude.bmp");

**// Добавление в коллекцию Controls формы.**

  Controls.Add(happyBox);

 }

}

Кроме свойства Image, нам будет интересно только свойство SizeMode, для которого используются значения перечня PiсtureBoxSizeMode. Этот тип используется для контроля того, как соответствующее изображение должно отображаться в рамках рабочего прямоугольника PictureBox. Здесь мы используем PictureBoxSizeMode.StretchImage, означающее то, что изображение следует растянуть на всю заданную типом PictureBox область (которая в данном случае имеет размеры 50×50 пикселей).

Следующей задачей является обработка событий MouseMove.MouseUр и MouseDown для члена-переменной PictureBox с помощью вполне стандартного синтаксиса обработки событий C#.

public MainForm() {

 …

**// Добавление обработчиков для ряда событий.**

 happyBox.MouseDown += new MouseEventHandler(happyBox\_MouseDown);

 happyBox.MouseUp += new MouseEventHandler(happyBox\_MouseUp);

 happyBox.MouseMove += new MouseEventHandler(hарруВох\_MouseMove);

 Controls.Add(happyBox);

 InitializeComponent();

}

Обработчик событий MouseDown сохраняет поступающие на вход значения координат (х, у) местоположения указателя в двух членах-переменных (oldX и oldY) для использования в дальнейшем, а также устанавливает значение true (истина) для члена-переменной (isDragging) типа System.Boolean, когда происходит перетаскивание. Добавьте эти члены-переменные в форму и реализуйте обработчик события MouseDown так, как предлагается ниже.

private void happyBox\_MouseDown(object sender, MouseEventArgs e) {

 isDragging = true;

 oldX = e.X;

 oldY = e.Y;

}

Обработчик события MouseMove просто изменяет местоположение PictureBox (с помощью свойств Тор и Left), в зависимости от сдвига положения указателя по сравнению со значениями, полученными при обработке события MouseDown.

private void happyBox\_MouseMove(object sender, MouseEventArgs e) {

 if (isDragging) {

**// Необходимо для вычисления нового значения Y в зависимости**

**// от того, где была нажата кнопка мыши.**

  happyBox.Top = happyBox.Top + (e.Y – oldY);

**// То же для X (используя в качестве основы oldX)**.

  happyBox.Left = happyBox.Left + (e.X – oldX);

 }

}

Обработчик события MouseUp устанавливает для isDragging значение false (ложь), чтобы сигнализировать об окончаний операции перетаскивания. Кроме того, если событие MouseUp происходит в тот момент, когда PictureBox содержится в пределах отображаемого средствами GDI+ Rectangle, мы будем считать, что пользователь победил в этой (очень примитивной) игре. Сначала добавьте в класс Form член Rectangle (с именем dropRect и заданными размерами).

public partial class MainForm: Form {

 private PictureBox happyBox = new PictureBox();

 private int oldX, oldY;

 private bool isDragging;

 private Rectangle dropRect = new Rectangle(100, 100, 140, 170);

 …

}

Обработчик события MouseUp теперь можно реализовать так.

private void happyBox\_MouseUp(object sender, MouseEventArgs e) {

 isDragging = false;

**// Находится ли указатель внутри заданного прямоугольника?**

 if (dropRect.Contains(happyBox.Bounds)) MessageBox.Show("Вы победили!", "Этот сладкий вкус умения…");

}

Наконец, в рамках обработки события Paint нужно отобразить в форме прямоугольную область (заданную переменной dropRect).

private void MainForm\_Paint(object sender, PaintEventArgs e) {

**// Отображение целевого прямоугольника.**

 Graphics g = e.Graphics;

 g.FillRectangle(Brushes.AntiqueWhite, dropRect);

**// Вывод инструкции.**

 g.DrawString("Тащите этого парня сюда!", new Font("Times New Roman", 25), Brushes.Red, dropRect);

}

Запустив свое приложение, вы увидите окно, подобное показанному на рис. 20.19.

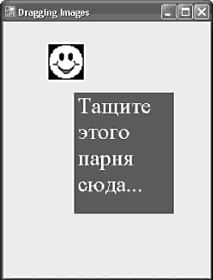


Рис. 20.19. Увлекательная игра "Счастливый пижон"

Если вы сделаете все, что требуется для победы в игре, вы увидите окно "восхваления", показанное на рис. 20.20.

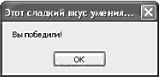


Рис 20.20. У вас железные нервы!

**Исходный код.**Проект DraggingImages размещен в подкаталоге, соответствующем главе 20.

**Проверка попадания в область изображения**

Проверить попадание в область типа, производного от Control (например, типа PictureBox очень просто, поскольку такой тип может сам отвечать на события мыши. Но что делать в том случае, когда нужно выполнять проверку попадания в область геометрического шаблона, отображенного на поверхности формы?

Для иллюстрации соответствующего процесса давайте снова рассмотрим предыдущее приложение BasicImages, чтобы наделить его некоторыми дополнительными возможностями. Нашей целью является выявление того, что пользователь щелкнул на одном из трех изображений. Выяснив, на каком именно изображении был выполнен щелчок, мы с помощью изменения свойства Text формы выделяем это изображение рамкой шириной в 5 пикселей.

Первым шагом должно быть определение нового множества членов-переменных типа Form, представляющих объекты Rectangle, для которых будет выполняться регистрация события MouseDown. При наступлении такого события нужно программно выяснить, находятся ли поступающие координаты (x, y) в рамках границ объектов Rectangle, используемых для визуализации объектов Image. Выяснив, что пользователь щелкнул на изображении, мы должны установить приватную булеву переменную (isImageClicked) равной true (истина) и указать, какое изображение было выбрано, используя для этого другую переменную и соответствующее значение из пользовательского перечня ClickedImage, определенного следующим образом.

enum ClickedImage {

 ImageA, ImageB, ImageC

}

С учетом сказанного, вот как может выглядеть исходная модификация нашего класса формы.

public partial class MainForm: Form {

 private Bitmap[] myImages = new Bitmap(3];

 private Rectangle[] imageRects = new Rectangle[3];

 private bool isImageClicked = false;

 ClickedImage imageClicked = ClickedImage.ImageA;

 public MainForm() {

  …

**// Установка прямоугольников.**

  imageRects[0] = new Rectangle(10, 10, 90, 90);

  imageRects[1] = new Rectangle(10, 110, 90, 90);

  imageRects[2] = new Rectangle (10, 210, 90, 90);

 }

 private void MainForm\_MouseDown(object sender, MouseEventArgs e) {

**// Получение координат (х, у) щелчка.**

  Point mousePt = new Point(e.X, e.Y);

**// Проверка попадания указателя в любой из прямоугольников.**

  if (imageRects[0].Contains(mousePt)) {

   isImageClicked = true;

   imageClicked = ClickedImage.ImageA;

   this.Text = "Вы щелкнули на изображении А";

  } else if (imageRects[1].Contains(mousePt)) {

   isImageClicked = true;

   imageClicked = Clickedlmage.ImageB;

   this.Text = "Вы щелкнули на изображении В";

  } else if (imageRects[2].Contains(mousePt)) {

   isImageClicked = true;

   imageClicked = ClickedImage.ImageC;

   this.Text = "Вы щелкнули на изображении C";

  } else { **// Попадания не обнаружено, использовать умолчания.**

   isImageClicked = false;

   this.Text = "Проверка попаданий в зону изображения";

  }

**// Обновление области клиента.**

  Invalidate();

 }

}

Обратите внимание на то, что при последней проверке член-переменная isImagеCliсked устанавливается равной false (ложь), поскольку пользователь не выполнил щелчка ни одном из трех изображений. Это важно, если вы хотите удалить контур у ранее выделенного изображения. После проверки всех элементов область клиента обновляется. Вот как выглядит модифицированный обработчик Paint.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

**// Визуализация изображений.**

 …

**// Прорисовка контура (при щелчке в соответствующем месте)**

 if (isImageClicked == true) {

  Pen outline = new Pen(Color.Tomato, 5);

  switch (imageClicked) {

  case ClickedImage.ImageA:

   g.DrawRectangle(outline, imageRects[0]);

   break;

  case Clickedlmage.ImageB:

   g.DrawRectangle(outline, imageRects[1]);

   break;

  case ClickedImage.ImageC:

   g.DrawRectangle(outline, imageRects[2]);

   break;

  default:

   break;

  }

 }

}

В данный момент вы уже можете запустить свое приложение и убедиться в том, что контур появляется вокруг каждого изображения, на котором был выполнен щелчок (и что никакого контура не появляется, когда вы щелкаете за пределами этих изображений).

**Проверка попадания в область, отличную от прямоугольной**

Теперь давайте выясним, как выполнить проверку попадания в область, форма которой отличается от прямоугольника? Предположим, что вы обновили свое приложение так, что теперь в нем отображается геометрический шаблон неправильной формы, и при щелчке на этом шаблоне его тоже требуется выделить с помощью контура (рис. 20.21).

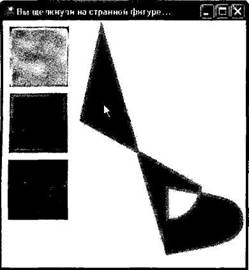


Рис. 20.21. Проверка попадания в многоугольники

Этот геометрический образ был создан на форме с помощью метода FillPath() типа Graphics. Указанный метод получает на вход экземпляр объекта GraphicsPath, инкапсулирующий последовательность соединенных линий, кривых и строк. Добавление новых элементов в экземпляр GraphicsPath осуществляется с помощью последовательности связанных методов Add, как описывается в табл. 20.9.

**Таблица 20.9.**Связанные методы Add класса GraphicsPath

|  |  |
| --- | --- |
| **Методы** | **Описание** |
| AddArc() | Добавляет к имеющейся фигуре эллиптическую дугу |
| AddBezier() AddBeziers() | Добавляет к имеющейся фигуре кубическую кривую Безье (или множество кривых Безье) |
| AddClosedCurve() | Добавляет к имеющейся фигуре замкнутую кривую |
| AddCurve() | Добавляет к имеющейся фигуре кривую |
| AddEllipse() | Добавляет к имеющейся фигуре эллипс |
| AddLine() AddLines() | Добавляет к имеющейся фигуре сегмент линии |
| AddPath() | Добавляет к имеющейся фигуре указанный GraphicsPath |
| AddPie() | Добавляет к имеющейся фигуре сектор круга |
| AddPolygon() | Добавляет к имеющейся фигуре многоугольник |
| AddRectangle() AddRectangles() | Добавляет к имеющейся фигуре прямоугольник (или несколько прямоугольников) |
| AddString() | Добавляет к имеющейся фигуре текстовую строку |

Укажите using System.Drawing.Drawing2D и добавьте новый член GraphicsPath в класс Form. В рамках конструктора формы постройте множество элементов, представляющих соответствующую траекторию.

public partial class MainForm: Form {

 GraphicsPath myPath = new GraphicsPath();

 public MainForm() {

**// Создание нужного пути.**

  myPath.StartFigure();

  myPath.AddLine(new Point(150, 10), new Point(120, 150));

  myPath.AddArc(200, 200, 100, 100, 0, 90);

  Point point1 = new Point(250, 250);

  Point point2 = new Point(350, 275);

  Point point3 = new Point (350, 325);

  Point point4 = new Point(250, 350);

  Point[] points = {point1, point2, point3, point4};

  myPath.AddCurve(points);

  myPath.CloseFigure();

  …

 }

}

Обратите внимание на вызовы StartFigure() и CloseFigure(). При вызове StartFigure() вы можете вставить новый элемент в траекторию, которую вы строите. Вызов CloseFigure() закрывает имеющуюся фигуру и начинает новую (если это требуется). Также следует знать, что в том случае, когда фигура содержит последовательность соединенных линий и кривых (как в случае с экземпляром myPath), цикл завершается путем соединения конечной и начальной точек с помощью линий. Сначала добавьте в перечень ClickedImage дополнительное имя StrangePath.

enum ClickedImage {

 ImageA, ImageB,

 ImageC, StrangePath

}

Затем обновите имеющийся обработчик события MouseDown, чтобы проверить присутствие указателя мыши в границах GraphicsPath. Как и для типа Region, это можно сделать с помощью члена IsVisible().

protected void OnMouseDown(object sender, MouseEventArgs e) {

**// Получение значений (х, у) для щелчка мыши.**

 Point mousePt = new Point(e.X, e.Y);

 …

 else if(myPath.IsVisible(mousePt)) {

  isImageClicked = true;

  imageClicked = ClickedImage.StrangePath;

  this.Text = "Вы щелкнули на странной фигуре…";

 }

 …

}

Наконец, измените обработчик Paint, как предлагается ниже.

private void MainForm\_Paint(object sender, PaintEventArgs e) {

 Graphics g = e.Graphics;

**// Рисование фигуры.**

 g.FillPath(Brushes.Sienna, myPath);

**// Рисование контура (при щелчке на соответствующей фигуре)**

 if (isImageClicked == true) {

  Pen outline = new Pen(Color.Red, 5);

  switch(imageClicked) {

   …

  case ClickedImage.StrangePath:

   g.DrawPath(outline, myPath);

   break;

  default:

   break;

  }

 }

}

**Исходный код.**Проект HitTestinglmages размещен в подкаталоге, соответствующем главе 20.

**Формат ресурсов .NET**

До этого момента все наши приложения, использующие внешние ресурсы (например, файлы изображений), требовали, чтобы загружаемые файлы находились в каталоге приложения. Поэтому для загрузки этих файлов мы использовали их непосредственные имена.

**// Загрузка изображений в объекты.**

bMapImageA = new Bitmap("imageA.bmp");

bMapImageB = new Bitmap("imageB.bmg");

bMapImageC = new Bitmap("imageC.bmp");

Такая программная логика требует, чтобы каталог приложения содержал три файла с именами imageA.bmp, imageB.bmp и imageС.bmp, иначе в среде выполнения будет сгенерировано соответствующее исключение.

Вы можете помнить из главы 11, что компоновочный блок представляет собой коллекцию типов и, необязательно, ресурсов. В этой связи нашей заключительной темой обсуждения в этой главе будет выяснение того, как выполнить привязку внешних ресурсов (например, файлов изображений и строк) непосредственно к компоновочному блоку. Тогда ваш двоичный блок .NET будет истинно самодостаточным. На элементарном уровне объединение внешних ресурсов в компоновочном блоке .NET предполагает выполнение следующих шагов.

1. Создание файла \*.resx в котором задаются пары имен и значений для каждого ресурса приложения в формате XML-представления данных.

2. Использование утилиты командной строки resgen.exe для преобразования XML-файла \*.resx в двоичный эквивалент (файл \*.resources).

3. Использование флага /resource компилятора C# для того, чтобы встроить двоичный файл \*.resources в компоновочный блок.

Как вы можете догадаться, в Visual Studio 2006 эти шаги автоматизированы. Чуть позже вы узнаете, как указанная среда разработки может вам в этом помочь. А пока давайте выясним, как сгенерировать и встроить ресурсы .NET в командной строке.

**Пространство имен System.Resources**

Ключом к пониманию формата ресурсов .NET является понимание типов, определенных в пространстве имен System.Resources. Соответствующее множество типов обеспечивает программные средства чтения и записи файлов \*.resx*(*в формате XML) и \*.resources (в двоичном формате), а также получения ресурсов, встроенных в компоновочный блок. Описания базовых типов этого пространства имен предлагаются в табл. 20.10.

**Таблица 20.10.**Члены пространства имен System.Resources

|  |  |
| --- | --- |
| **Члены** | **Описание** |
| ResourceReader ResourceWriter | Позволяют читать и записывать данные двоичных файлов |
| ResXResourceReader ResXResourceWriter | Позволяют читать и записывать данные XML-файлов \*.resx |
| ResourceManager | Позволяет программно получить встроенные ресурсы данного компоновочного блока |

**Создание файла \*.resx программными средствами**

Как было отмечено выше, файл \*.resx содержит XML-данные, представляющие пары имен и значений для каждого ресурса приложения. Класс ResXResourceWriter предлагает набор членов, с помощью которых вы можете создать файл \*.resx, добавить в него двоичные и строковые ресурсы и сохранить их. Для примера мы создадим простое приложение (ResXWriter), которое будет генерировать файл \*.resx, содержащий информацию, необходимую для загрузки файла happyDude.bmp (впервые упомянутого в примере DraggingImages) и некоторого строкового ресурса. Графический интерфейс пользователя будет образован единственным типом Button (рис. 20.22).



Рис. 20.22. Приложение ResX

Обработчик события Click для Button добавляет happyDude.bmp и строковый ресурс в файл \*.resx, который сохраняется на локальном диске C.

private void btnGenResX\_Click(object sender, EventArgs e) {

**// Создание объекта, записывающего данные resx,**

**// и указание файла для записи.**

 ResXResourceWriter w = new ResXResourceWriter(@"C:\ResXForm.resx");

**// Добавление изображения и строки.**

 Image i = new Bitmap ("happyDude.bmp");

 w.AddResource("happyDude", i);

 w.AddResource("welcomeString", "Приветствие формата ресурсов");

**// Фиксация файла.**

 w.Generate();

 w.Close();

}

Здесь наибольший интерес представляет собой член ResXResourceWriter.AddResource(). Этот метод является перегруженным, чтобы вы имели возможность вставить как двоичные данные (такие, как изображение happyDude.bmp), так и текстовые (наподобие указанной выше строки). Обратите внимание на то, что каждый из вариантов предполагает использование двух параметров: имени соответствующего ресурса в файле \*.resx и непосредственно данных. Метод Generate() фиксирует информацию в файле. В результате вы получаете XML-описание ресурсов изображения и строки. Для проверки откройте новый файл ResXForm.resx с помощью текстового редактора (рис. 20.23).

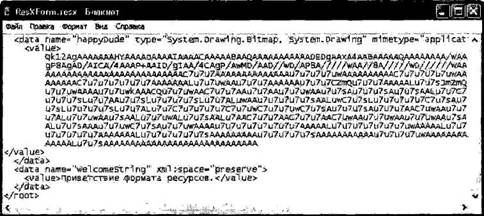


Рис. 20.23. XML-представление \*.resx

**Создание файла \*.resources**

Имея файл \*.resx, вы можете использовать утилиту resgen.exe, чтобы сгенерировать двоичный эквивалент этого файла. Для этого откройте окно командной строки Visual Studio 2005, перейдите в корневой каталог вашего диска C и выполните следующую команду.

resgen resxform.resx resxform.resources

Теперь вы можете открыть новый файл \*.resources с помощью Visual Studio 2005, чтобы увидеть данные в двоичном формате (рис. 20.24).



Рис. 20.24. Двоичное представление \*.resources

**Добавление файла \*.resources в компоновочный блок .NET**

Теперь можно встроить файл \*.resources в компоновочный блок.NET, используя опцию /resources компилятора командной строки C#. Для примера скопируйте соответствующие файлы Program.cs, Form1.cs и Form1.Designer.cs в корневой каталог вашего диска C, откройте окно командной строки Visual Studio 2005 и выполните следующую команду.

csc /resource:resxform.resources /r:System.Drawing.dll \*.cs

Если открыть созданный в результате этого компоновочный блок с помощью ildasm.exe, вы увидите, что манифест соответствующим образом обновлен, как показано на рис. 20.25.

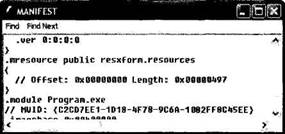


Рис. 20.25. Встроенные ресурсы

**Работа с ResourceWriter**

В предыдущем примере мы использовали тип ResXResourceWriter, чтобы сгенерировать XML-файл, содержащий пары имен и значений для каждого ресурса приложения. Полученный в результате файл \*.resx был затем обработан утилитой resgen.exe. Наконец, с помощью флага /resource компилятора C# файл \*. resources был встроен в компоновочный блок. Но дело в том. что создавать файл \*.resx совсем не обязательно (хотя наличие XML-представления ресурсов может оказаться полезным, поскольку оно вполне пригодно для анализа). Если вам не нужен файл \*.resx, вы можете использовать тип ResourceWriter, с помощью которого двоичный файл \*.resources можно создать непосредственно.

private void GenerateResourceFile() {

**// Создание нового файла \*.resources**.

 ResourceWriter rw;

 rw = new ResourceWriter(@"C:\myResources.resources");

**// Добавление одного изображения и одной строки.**

 rw.AddResource("happyDude", new Bitmap("happyDude.bmp"));

 rw.AddResource("welcomeString", "Приветствие формата ресурсов.");

 rw.Generate();

 rw.Close();

}

Полученный файл \*.resources можно добавить в компоновочный блок, используя опцию /resources компилятора.

csc /resource:myresources.resources \*.cs

**Исходный код.**Проект ResXWriter размещен в подкаталоге, соответствующем главе 20.

**Генерирование ресурсов в Visual Studio 2005**

С файлами \*.resx и \*.resources можно, конечно, работать вручную в командной строке, но Visual Studio 2005 предлагает средства автоматизации создания и встраивания ресурсов в проект. Для примера создайте новое приложение Windows Forms, назвав его MyResourcesWinApp. Открыв для созданного проекта окно Solution Explorer, вы увидите, что теперь с каждой формой в вашем приложении автоматически связывается файл \*.resx (рис. 20.26).

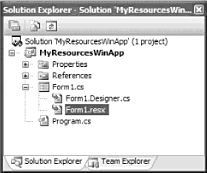


Рис. 20.26. Автоматическое генерирование файлов \*.resx в Visual Studio 2005

При добавлении вами ресурсов (например, изображений в PictureBox) с помощью визуальных инструментов проектирования этот файл \*.resx будет изменяться автоматически. Более того, вы *не должны*вручную обновлять этот файл, чтобы указать пользовательские ресурсы, поскольку Visual Studio 2005 регенерирует этот файл при каждой компиляции проекта. Чтобы гарантировать правильную структуру этого файла, лучше всего позволить среде разработки самой управлять файлом \*.resx формы.

Если вы захотите предложить пользовательский набор ресурсов, не связанный непосредственно с данной формой, просто добавьте новый файл \*.resx (в нашем примере это файл MyCustomResources.resx), выбрав команду Project→Add New Item из меню (рис. 20.27).

Если открыть новый файл \*.resx, соответствующий редактор позволит вставить необходимые строковые данные, изображения, звуковые фрагменты и другие ресурсы. Крайнее слева раскрывающееся меню позволит выбрать вид ресурса, который вы хотите добавить. Сначала добавьте новый строковый ресурс с именем WelcomeString, чтобы с его помощью задать то сообщение, которое вы хотите видеть (рис. 20.28).

Теперь добавьте изображение happyDude.bmp, сначала выбрав Images из крайнего меню, после этого – Add Existing File (рис. 20.29), а затем – указав файл happyDude.bmp в появившемся окне загрузки файла.

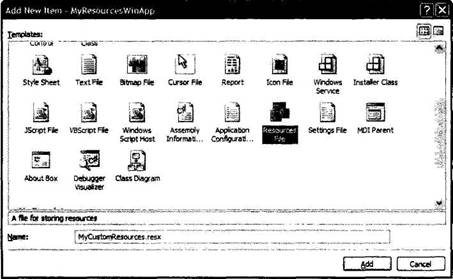


Рис. 20.27. Добавление нового файла \*.resx

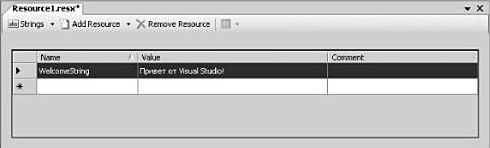


Рис. 20.28. Добавление нового строкового ресурса с помощью редактора файлов \*.resx



Рис. 20.29. Добавление нового ресурса \*.bmp с помощью редактора файлов \*.resx

После этого вы обнаружите, что файл \*.bmp скопирован в каталог вашего приложения. Выбрав пиктограмму happyDude в редакторе \*.resx и используя возможности свойства Persistеnce, можно потребовать, чтобы данное изображение было непосредственно встроено в компоновочный блок, а не задавалось ссылкой на внешний файл (рис. 20.30).



Рис. 20.30. Встраивание указанных ресурсов

Теперь окно Solution Explorer предлагает новую папку под названием Resources, которая содержит все элементы, встроенные в компоновочный блок. Легко догадаться, что при открытии ресурса Visual Studio 2005 запускает соответствующий редактор. В любом случае, если вы теперь скомпилируете свое приложение, указанные вами строка и изображение будут встроены в компоновочный блок.

**Чтение ресурсов программными средствами**

Теперь, когда вы понимаете суть процесса встраивания ресурсов в компоновочный блок (с помощью csc.exe или Visual Studio 2005), нужно выяснить, как с помощью типа ResourceManager программно прочитать соответствующую информацию для использования в программе. Для этого добавьте в свою форму элементы Button и PictureBox (рис. 20.31).



Рис. 20.31. Обновленный интерфейс пользователя

Теперь обработайте событие Click для Button. Добавьте в обработчик указанного события следующий программный код.

**// Не забудьте указать 'using'**

**// для System.Resources и System.Reflection!**

 private void btnGetStringData\_Click(object sender, EventArgs e) {

**// Создание менеджера ресурсов.**

 ResourceManager rm = new ResourceManager("MyResourcesWinApp.MyCustomResources", Assembly.GetExecutingAssembly());

**// Получение встроенной строки (с учетом регистра!)**

 MessageBox.Show(rm.GetString("WelcomeString"));

**// Получение встроенного изображения (с учетом регистра!)**

 myPictureBox.Image = (Bitmap)rm.GetObject("HappyDude");

**// Необходимая 'уборка'.**

 rm.ReleaseAllResources();

}

Обратите внимание на то, что первый аргумент конструктора ResourceManager представляет собой полное имя файла \*.resx (но без расширения файла). Второй параметр является ссылкой на компоновочный блок, содержащий соответствующий встроенный ресурс (в данном случае это текущий компоновочный блок). После создания ResourceManager вы можете вызвать GetString() или GetObject(), чтобы извлечь встроенные данные. Если вы запустите приложение и щелкнете на его кнопке, вы увидите, что извлеченные из компоновочного блока строковые данные отобразятся в MessageBox, а данные изображения – в PictureBox.

**Исходный код.**Проект MyResourcesWinApp размещен в подкаталоге, соответствующем главе 20.

На этом мы завершаем наш обзор возможностей GDI+ и пространств имен System.Drawing. Если вы заинтересованы в дальнейшем исследовании GDI+ (включая поддержку печати), обратитесь к книге Nick Symmonds, *GDI+ Programming in C# and VB.NET*(Apress, 2002).

**Резюме**

Аббревиатура GDI+ используется для обозначения ряда связанных пространств имен .NET, используемых для визуализации графических образов на поверхности производных от Control типов. Значительная часть этой главы была посвящена выяснению того, как работать с базовыми объектными типами GDI+ – цветами, шрифтами, графическими изображениями, перьями и кистями, используемыми в совокупности с "всемогущим" типом Graphics. В процессе обсуждения были рассмотрены и некоторые другие возможности GDI+, такие как проверка попадания указателя мыши в заданную область окна и перетаскивание изображений.

В завершение главы был рассмотрен новый формат ресурсов .NET. Как выяснилось, файл \*.resx определяет ресурсы в виде XML с помощью пар имен и значений. Этот файл можно обработать с помощью утилиты resgen.exe, чтобы привести его к двоичному формату (\*.resources) и затем встроить соответствующие ресурсы в компоновочный блок. Наконец, тип ResourceManager обеспечивает простой способ программного извлечения встроенных ресурсов во время выполнения приложения.

**ГЛАВА 21. Использование элементов управления Windows Forms**

Эта глава представляет собой краткое руководство по использованию элементов управления, определенных в пространстве имен System.Windows.Forms. В главе 19 вы уже имели возможность поработать с некоторыми элементами управления, размещаемыми в главной форме: это MenuStrip, ToolStrip и StatusStrip. В этой главе мы рассмотрим различные типы, которые обычно размещают в пределах области клиента формы (это, например, Button, MaskedTextBox, WebBrowser, MonthCalendar, TreeView и т.п.). Рассмотрев базовые элементы пользовательского интерфейса, мы затем обсудим процесс создания пользовательских элементов управления Windows Forms, интегрируемых в среду разработки Visual Studio 2005.

После этого мы рассмотрим процесс построения пользовательских диалоговых окон, а также роль *наследования форм,*которое позволяет создавать иерархии связанных типов Form. В завершение главы обсуждается возможность *стыковки*и *закрепления*элементов графического интерфейса и выясняется роль типов FlowControlPanel и TableControlPanel, предлагаемых в .NET 2.0.

**Элементы управления Windows Forms**

Пространство имен System.Windows.Forms содержит ряд типов, представляющих наиболее часто используемые элементы графического интерфейса, которые обеспечивают поддержку взаимодействия с пользователем в приложении Windows Forms. Многие элементы управления из тех, с которыми вы будете работать ежедневно (такими, например, являются Button, TextBox и Label), интуитивно совершенно понятны, но чтобы работать с другими, более "экзотическими" элементами управления и компонентами (например, с TreeView, ErrorProvider и TabControl), требуются дополнительные пояснения.

Из главы 19 вы узнали, что тип System.Windows.Forms.Control является базовым классом для всех таких элементов. Напомним, что Control обеспечивает возможность обрабатывать события мыши и клавиатуры, задавать физические размеры и позицию элементов в форме с помощью различных свойств (Height, Width, Left, Right, Location и т.п.), манипулировать цветами фона и переднего плана, задавать активный шрифт/курсор и т.д. Кроме того, базовый тип Control определяет члены, контролирующие возможность закрепления и поведение стыковки элемента (объяснения по поводу указанных возможностей будут даны в тексте этой главы позже).

При изучении материала данной главы помните о том, что рассматриваемые здесь элементы управления наследуют большинство своих функциональных возможностей от базового класса Control. Поэтому мы сосредоточимся (более или менее) на уникальных членах этих элементов. Следует понимать, что эта глава не пытается полностью описать абсолютно все члены абсолютно всех элементов управления (это задача документации .NET Framework 2.0 SDK). Однако я уверен, что после завершения изучения материала этой главы у вас не возникнет проблем в понимании тех элементов, которые здесь непосредственно описаны не были.

**Замечание.**В Windows Forms предлагается целый ряд элементов управления (DataGridView, BindingSource и т.д.), позволяющих отображать содержимое реляционных баз данных. Некоторые из этих элементов управления будут рассмотрены в главе 22 при обсуждении ADO.NET.

**Добавление элементов управления в форму вручную**

Независимо от того, какой именно элемент управления вы выбираете для размещения в форме, последовательность шагов, которые при этом выполняются, оказывается одной и той же. Во-первых, вы должны определить члены-переменные, представляющие эти элементы управления. Затем, в рамках конструктора формы (или вспомогательного метода, вызванного конструктором), вы должны настроить вид каждого элемента управления, используя доступные свойства, методы и события этого элемента. Наконец (и это самое важное), после установки элемента управления в исходное состояние, вы должны добавить этот элемент к внутренней коллекции элементов управления формы, используя унаследованное свойство Controls. Если вы не сделаете этот заключительный шаг, ваш элемент управления *не будет*видимым во время выполнения.

Чтобы рассмотреть процесс добавления элементов управления в форму, давайте начнем с построения типа Form без помощи мастеров, используя только текстовый редактор и компилятор командной строки C#. Создайте новый файл C# с именем ControlsByHand.cs и добавьте в этот файл следующий программный код, определяющий новый класс MainWindow.

using System;

using System.Drawing;

using System.Windows.Forms;

namespace ControlsByHand {

 class MainWindow: Form {

**// Члены-переменные элементов формы.**

  private TextBox firstNameBox = new TextBox();

  private Button btnShowControls = new Button();

  public MainWindow() {

**// Конфигурация формы.**

   this.Text = "Простые элементы управления";

   this.Width = 300;

   this.Height = 200;

   CenterToScreen();

**// Добавление в форму нового текстового окна.**

   firstNameBox.Text = "Привет";

   firstNameBox.Size = new Size(150, 50);

   firstNameBox.Location = new Point(10, 10);

   this.Controls.Add(firstNameBox);

**// Добавление в форму новой кнопки.**

   btnShowControls.Text = "Щелкните здесь";

   btnShowControls.Size = new Size(90, 30);

   btnShowControls.Location = new Point(10, 10);

   btnShowControls.BackColor = Color.DodgerBlue;

   btnShowControls Click += new EventHandler(btnShowControls\_Clicked);

   this Controls.Add(btnShowControls);

  }

**// Обработка события Click кнопки.**

  private void.btnShowControls\_Clicked(object sender, EventArgs e) {

**// Вызов ToString() для каждого элемента управления**

**// из коллекции Controls формы.**

   string ctrlInfo = "";

   foreach (Control c in this.Controls) {

    ctrlInfo += string.Format.("Элемент: {0}\n:", s.ToString());

   }

   MessageBox.Show(ctrlInfo, "Элементы управления, помещенные в форму");

  }

 }

}

Добавьте в пространство имен ControlsByHand еще один класс, реализующий метод Main() программы.

class Program {

 public static void Main(string[] args) {

  Application.Run(new MainWindow());

 }

}

Теперь скомпилируйте полученный файл C# в командной строке, используя следующую команду.

csc /target:winexe \*.cs

Запустив приложение и щелкнув на кнопке формы, вы увидите окно сообщения, в котором будет представлен список всех элементов, помещенных в эту форму (рис. 21.1).



Рис. 21.1. Взаимодействие с коллекцией элементов управления формы

**Тип Control.ControlCollection**

Процедура добавления нового элемента в форму исключительно проста, но свойство Controls требует дополнительного обсуждения. Это свойство возвращает ссылку на вложенный класс с именем ControlCollection, определенный в рамках класса Control. Вложенный тип ControlCollection содержит все элементы управления, помещенные в данную форму. Вы можете получить ссылку на эту коллекцию, чтобы "рассмотреть список" дочерних элементов формы.

**// Получение доступа к вложенной коллекции ControlCollection формы.**

Control.ControlCollection coll = this.Controls;

Имея ссылку на эту коллекцию, вы можете манипулировать ее содержимым, используя члены, описания которых предлагаются в табл. 21.1.

**Таблица 21.1.**Члены ControlCollection

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Add() AddRange() | Используются для добавления в коллекцию нового производного от Control типа (или массива типов) |
| Clear() | Удаляет все элементы из коллекции |
| Count | Возвращает число элементов, имеющихся в коллекции |
| GetEnumerator() | Возвращает интерфейс IEnumerator для данной коллекции |
| Remove() RemoveAt() | Используются для удаления элемента из коллекции |

Ввиду того, что форма поддерживает коллекцию элементов управления, в Windows Forms очень просто динамически создавать, удалять или как-то иначе обрабатывать визуальные элементы. Предположим, например, что вы хотите отключить все типы Button в данной форме (или выполнить иное подобное действие, например, изменить цвет фона всех TextBox). Для этого можно использовать ключевое слово is C#, чтобы обнаружить нужные элементы и соответственно изменить их состояние.

private void DisableAllButtos() {

 foreach (Control с in this.Controls) {

  if (c is Button) ((Button)c).Enabled = false;

 }

}

**Исходный код**. Проект ControlsByHand размещен в подкаталоге, соответствующем главе 21.

**Добавление элементов управления в форму в Visual Studio 2005**

Теперь, когда вы понимаете суть процесса добавления элементов управления в форму, давайте посмотрим, как Visual Studio 2005 может автоматизировать этот процесс. Создайте новый проект Windows Application, выбрав дня него произвольное имя, поскольку этот проект будет предназначен только для тестирования. Аналогично рассмотренному в главе 19 случаю создания меню, панелей инструментов и строк состояния, когда среда IDE выполняла автоматическое добавление подходящей переменной в файл \*.Designer.cs, те же действия выполняются средой разработки и при перетаскивании в окно проектирования формы любого другого элемента управления. Точно так же при изменении внешнего вида элемента с помощью окна свойств IDE выполняется соответствующие изменения программного кода члена-функции InitializeComponent() (также размещенного в файле \*.Designer.cs).

Замечание. Напомним, что окно свойств, если щелкнуть в нем на кнопке с пиктограммой молнии, позволяет обработать события элемента управления. Нужно выбрать элемент из раскрывающегося списка и указать имя метода, который должен вызываться для интересующих вас событий (или просто выполнить двойной щелчок на имени события, чтобы сгенерировать обработчик события с именем, предлагаемым по умолчанию).

Добавьте в окно проектирования формы типы TextBox (текстовое окно) и Button (кнопка). Обратите внимание на то, что при изменении положения элемента управления в окне проектирования формы Visual Studio 2005 предлагаются визуальные подсказки, касающиеся размещения и выравнивания этого элемента (рис. 21.2).

После размещения Button и Textbox в окне проектирования формы рассмотрите программный код, сгенерированный в методе InitializeComponent(). Вы обнаружите, что соответствующие типы автоматически были созданы с помощью new и добавлены в коллекцию ControlCollection формы (в дополнение к установкам, которые вы, возможно, добавили с помощью окна свойств).

private void InitializeComponent() {

 this.btnMyButton = new System.Windows.Forms.Button();

 this.txtMyTextBox = new System.Windows.Forms.TextBox();

 …

**// MainWindow**

**//**

 …

 this.Controls.Add(this.txtMyTextBox);

 this.Controls.Add(this.btnMyButton);

 …

}

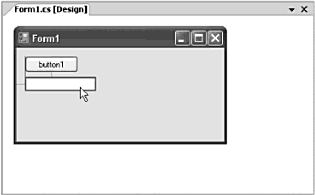


Рис. 21.2. Визуальные подсказки по поводу выравнивания и размещения элементов управления в форме

Как видите, такие инструменты, как Visual Studio 2005, во многом избавляют вас от необходимости вводить программный код вручную (возможно, избавляя и от боли в пальцах). Итак, в результате ваших действий в окне проектирования формы среда разработки автоматически модифицирует InitializeComponent(). Но вы также можете конфигурировать элементы управления и непосредственно в программном коде (в конструкторах, обработчиках событий, вспомогательных функциях и т.д.), поскольку задачей InitializeComponent() является создание начального состояния элементов интерфейса. Тем не менее, если вы хотите упростить себе жизнь, позвольте Visual Studio 2005 поддерживать InitializeComponent() автоматически, поскольку средства проектирования формы могут проигнорировать или переписать изменения, сделанные вами в рамках этого метода.

**Работа с базовыми элементами управления**

Пространство имен System.Windows.Forms определяет множество "базовых элементов управления", которые можно видеть во многих типичных окнах (это кнопки, надписи, текстовые окна, переключатели и т.д.). Вы, наверное, уже знаете об основных возможностях соответствующих типов, но давайте рассмотрим некоторые наиболее интересные особенности следующих базовых элементов пользовательского интерфейса:

• Label (надпись), TextBox (текстовое окно) и MaskedTextBox (маскированное текстовое окно);

• Button (кнопка);

• CheckBox (кнопка с независимой фиксацией), RadioButton (кнопка с зависимей фиксацией) и GroupBox (групповой блок);

• CheckedListBox (окно отмечаемого списка), ListBox (окно списка) и ComboBox (комбинированное окно).

Освоив работу с этими элементами управления, производными от типа Control, мы с вами обратим внимание на более "экзотические" элементы, такие как MonthCalendar, TabControl, TrackBar, WebBrowser и т.д.

**Элемент Label**

Элемент управления Label (надпись) может содержать информацию, доступную только для чтения (текст или изображение), например, для того, чтобы пояснить пользователю роль и возможности использования остальных элементов управления, помещенных в форму. Предположим, вы создали в Visual Studio 2005 новый проект Windows Forms с именем LabelsAndTextBoxes. В рамках полученного экземпляра типа Form определите метод с именем CreateLabelControl(), который сначала создает и конфигурирует тип Label, а затем добавляет его в коллекцию элементов управления формы,

private void CreateLabelControl() {

**// Создание и конфигурация Label.**

 Label lblInstructions = new Label();

 lblInstructions.Name = "lblInstructions";

 lblInstructions.Text = ''Введите значения во все окна текста";

 lblInstructions.Font = new Font("Times New Roman", 9.75F, FontStyle.Bold);

 lblInstructions.AutoSize = true;

 lblInstructions.Location = new System.Drawing.Point(16, 13);

 lblInstructions.Size = new System.Drawing.Size(240, 16);

**// Добавление в коллекцию элементов управления формы.**

 Controls.Add(lblInstructions);

}

Чтобы увидеть соответствующую подсказку в верхней части главного окна формы, поместите вызов этой вспомогательной функции в конструктор формы.

public MainWindow() {

 InitializeComponent();

 CreateLabelControl();

 CenterToScreen();

}

В отличие от большинства других элементов, элементы управления Label не могут получать фокус ввода при переходах по клавише табуляции. Однако в .NET 2.0 для любого элемента управления Label можно создать *мнемонические клавиши* установив для свойства UseMnemonic значение true (именно это значение устанавливается для данного свойства по умолчанию). После этого в свойстве Text надписи можно (с помощью символа амперсанда &) определить комбинацию клавиш для перехода к соответствующему элементу управления.

**Замечание.**Порядок переходов по табуляции подробнее будет обсуждаться позже, а пока что достаточно заметить, что порядок данного элемента управления при переходах по клавише табуляции устанавливается с помощью свойства TabIndex. По умолчанию значение TabIndex элемента управления соответствует порядку, в котором элементы добавлялись в окно проектирования формы. Поэтому, если вы сначала добавите Label, а затем – Textbox, то для Label значение TabIndex будет установлено равным 0, а для Textbox – равным 1.

Для примера с помощью окна проектирования формы постройте пользовательский интерфейс, состоящий из трех Label и трех Textbox (рис. 21.3). Не забудьте оставить свободное место в верхней части формы для отображения элемента Labels динамически создаваемого в методе CreateLabelControl(), и обратите внимание на то, что здесь каждая надпись содержит подчеркнутую букву. Так выделяются буквы, которые в значении свойства Text надписи были помечены знаком &. Вы, наверное, знаете, что помеченные знаком & символы обеспечивают возможность активизации элемента пользователем с помощью выбора комбинации клавиш ‹Alt+ *помеченный символ›.*

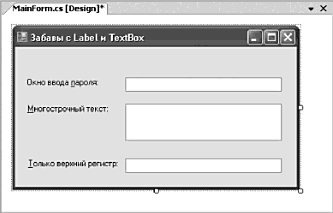


Рис. 21.3. Назначение мнемоник элементам управления Label

Запустив проект, вы сможете перейти к любому из имеющихся элементов TextBox, используя ‹Alt+n›, ‹Alt+M› или ‹Alt+T›.

**Элемент TextBox**

В отличие от элемента управления Label, элемент управления TextBox (текстовое окно) обычно не является доступным только для чтения (хотя его можно сделать таким, если установить для свойства ReadOnly значение true) и часто используется как раз для того, чтобы позволить пользователю ввести текстовые данные для обработки. Тип TextBox можно настроить на поддержку одной строки текста или множества строк, его можно настроить на маскировку вводимых символов (например, с помощью звездочки, \*) и в случае многострочного ввода этот элемент может содержать полосы прокрутки. Вдобавок к возможностям, унаследованным от базовых классов, TextBox определяет несколько своих интересных свойств (табл. 21.2).

Таблица 21.2. Свойства TextBox

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| AcceptsReturn | Читает или задает значение, являющееся индикатором необходимости перехода на новую строку при нажатии ‹Enter› в элементе управления TextBox, допускающем многострочный ввод (иначе нажатие ‹Enter› активизирует кнопку формы, используемую по умолчанию) |
| CharacterCasing | Читает или задает значение, указывающее необходимость изменения элементом управления TextBox регистра символов при их вводе |
| PasswordChar | Читает или задает символ, применяемый для маскировки вводимых символов в однострочном элементе управления TextBox, используемом для ввода паролей |
| ScrollBars | Читает или задает значение, указывающее необходимость наличии полос прокрутки в элементе управления TextBox, допускающем многострочный ввод |
| TextAlign | Читает или задает значение, соответствующее одному из значений перечня HorizontalAlignment и указывающее правила выравнивания текста в элементе управления TextBox |

Чтобы продемонстрировать некоторые возможности TextBox, поместите в форму три элемента управлений TextBox. Первый элемент TextBox (с именем txtPassword) следует настроить для ввода пароля, т.е. символы, вводимые в поле TextBox, не должны быть видимыми, а вместо них должны появляться символы, заданные значением свойства PasswordChar.

Второй элемент TextBox (с именем txtMultiline) – это окно многострочного текста, которое должно допускать обработку нажатия.клавиши ввода и отображать вертикальную полосу прокрутки, когда введенный текст не умещается в рамках видимого пространства TextBox. Наконец, третий элемент TextBox (с именем txtUppercase) будет настроен на перевод введенных символьных данных в верхний регистр.

Сконфигурируйте каждый элемент TextBox соответствующим образом с помощью окна свойств, используя в качестве руководства следующий фрагмент реализации InitializeComponent().

private void InitializeComponent() {

 …

**// txtPassword**

**//**

 this.txtPassword.PasswordChar = '\*';

 …

**// txtMultiline**

**//**

 this.txtMultiline.Multiline = true;

 this.txtMultiline.Scrollbars = System.Windows.Forms.ScrollBars.Vertical;

 …

**// txtUpperCase**

**//**

 this.txtUpperCase.CharacterCasing = System.Windows.Forms.CharacterCasing.Upper;

 …

}

Свойству ScrollBars присваивается значение из перечня ScrollBars, который определяет следующие элементы.

public enum System.Windows.Forms.ScrollBars {

 Both, Horizontal, None, Vertical

}

Свойство CharacterCasing работает в паре с перечнем CharacterCasing, который определен так.

public enum System.Windows.Forms.CharacterCasing {

 Normal, Upper, Lower

}

Сейчас предположим, что мы поместили в форму кнопку Button (с именем btnDisplayData) и добавили для этой кнопки обработчик события Click (щелчок]. Наша реализация соответствующего метода просто отображает значения всех элементов TextBox в окне сообщения.

private void btnDisplayData\_Click(object sender, EventArgs e) {

**// Получение данных всех текстовых окон.**

 string textBoxData *=*";

 textBoxData += string.Format("MultiLine: {0}\n", txtMultiline.Text);

 textBoxData += string.Format("\nPassword: {0}\n", txtPassword.Text);

 textBoxData += string.Format("\nUppercase: {0}\n", txtUpperCase.Text);

**// Отображение полученных данных.**

 MessageBox.Show(textBoxData, "Вот содержимое, элементов TextBox");

}

На рис. 21.4 показан один из возможных вариантов ввода (обратите внимание на то. что вы должны нажать клавишу ‹Alt›. чтобы увидеть мнемоники надписей).



Рис. 21.4. Множество "воплощений" типа TextBox

На рис. 21.5 показан результат выполнения щелчка на элементе типа Button.

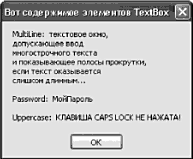


Рис. 21.5. Извлечение значений из объектов TextBox

**Элемент MaskedTextBox**

В .NET 2.0 предлагается также *маскированное*текстовое окно, которое позволяет задать последовательность символов, допустимую для принятия буфером ввода (это может быть номер социальной страховки, телефонный номер с кодом региона, почтовый индекс или что-то другое). Маска, с которой производится сравнение (она называется *шаблоном*или *выражением маски),* создается с помощью специальных маркеров, встроенных в строковый литерал. Созданное значение выражения маски присваивается свойству Mask. В табл. 21.3 даются описания некоторых масочных маркеров.

Таблица 21.3. Маркеры типа MaskedTextBox

|  |  |
| --- | --- |
| **Маркер** | **Описание** |
| 0 | Представляет наличие цифры (значения 0-9) |
| 9 | Представляет необязательную цифру или пробел |
| L | Представляет наличие буквы (в верхнем или нижнем регистре, A-Z) |
| ? | Представляет необязательную букву (в верхнем или нижнем регистре, A-Z) |
| , | Представляет разделитель тысяч |
| : | Представляет указатель места заполнения времени |
| / | Представляет указатель места заполнения даты |
| $ | Представляет символ денежной единицы |

Замечание. Символы, допустимые для использования с типом MaskedTextBox,*не вполне*соответствуют синтаксису регулярных выражений. Хотя .NET и предлагает специальные пространства имен для работы со стандартными регулярными выражениями (это пространства имен System.Text.RegularExpressions и System.Web.RegularExpressions), тип MaskedTextBox использует синтаксис, аналогичный синтаксису элементов управления COM в VB6.

Вдобавок к свойству Mask, тип MaskedTextBox предлагает члены, определяющие реакцию этого элемента управления на ввод пользователем некорректных данных. Например, BeepOnError (очевидно) заставит элемент управления сгенерировать звуковой сигнал, если ввод не соответствует маске, и некорректный символ к обработке допущен не будет.

Чтобы показать возможности использования MaskedTextBox, добавьте в имеющуюся у вас форму дополнительные элементы Label и MaskedTextBox. Вы, конечно, можете построить шаблон маски непосредственно в программном коде вручную, но в строке свойства Mask в окне свойств предлагается кнопка с многоточием, которая открывает диалоговое окно с целым рядом уже готовых масок (рис. 21.6).

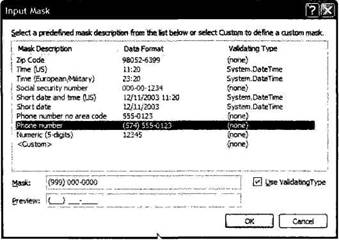


Рис. 21.6. Встроенные значения шаблонов маски для свойства Mask

Выберите подходящий шаблон (например, Phone number – телефонный номер), активизируйте свойство BeepOnError и снова выполните тестовый запуск программы. Вы обнаружите, что теперь вам не позволяется вводить никаких буквенных символов (в случае выбора маски Phone number).

Как и следует ожидать, элемент управления MaskedTextBox в цикле своего существования генерирует различные события, одно из которых – это событие MaskInputRejected, возникающее при вводе пользователем некорректных данных. Обработайте это событие, используя окно свойств, и обратите внимание на то, что вторым входным аргументом сгенерированного обработчика события является тип MaskInputRejectedEventArgs. Этот тип имеет свойство RejectionHint, содержащее краткое описание возникшей ошибки. Для проверки просто отобразите информацию об ошибке в строке заголовка формы.

private void txtMaskedTextBox\_MaskInputRejected(object sender, MaskInputRejectedEventArgs e) {

 this.Text = string.Format("Ошибка: {0}", e.RejectionHint);

}

Чтобы гарантировать, что эта ошибка не будет отображаться при вводе пользователем корректных данных, обработайте событие KeyDown для MaskedTextBox и реализуйте в обработчике события восстановление заголовка формы к значению, принятому по умолчанию.

private void txtMaskedTextBox\_KeyDown(object sender, KeyEventArgs e) {

 this.Text = "Забавы с Label и TextBox";

}

**Исходный код**. Проект LabelsAndTextBoxes размещен в подкаталоге, соответствующем главе 21.

**Элемент Button**

Задачей типа System.Windows.Forms.Button является "транспортировка" информации о выборе пользователя, обычно в ответ на щелчок кнопки мыши или нажатие клавиши пользователем. Класс Button (кнопка) получается непосредственно из абстрактного типа ButtonBase, обеспечивающего ряд ключевых возможностей поведения для всех производных типов (таких, как CheckBox, RadioButton и Button). В табл. 21.4 описаны некоторые базовые свойства ButtonBase.

Таблица 21.4. Свойства ButtonBase

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| FlatStyle | Возвращает или задает значение, соответствующее одному из элементов перечня FlatStyle и указывающее стиль внешнего вида элемента управления Button |
| Image | Указывает (необязательное) изображение, которое будет отображаться где-то в границах производного от ButtonBase типа. Напомним, Что класс Control определяет свойство BackgroundImage, которое используется для визуализации изображения на всей поверхности элемента |
| ImageAlign | Задает значение, соответствующее одному из элементов перечня ContentAlignment и указывающее правила выравнивания изображения на элементе управления Button |
| TextAlign | Возвращает или задает значение, соответствующее одному из элементов перечня ContentAlignment и указывающее правила выравнивания текста на элементе управления Button |

Свойство TextAlign типа ButtonBase сильно упрощает задачу позиционирования соответствующего текста. Чтобы позиционировать текст на поверхности Button, используйте значения перечня ContentAlignment (определенного в пространстве имен System.Drawing). Позже вы увидите, что этот же перечень можно использовать и при размещении на поверхности Button изображения.

public enum System.Drawing.ContentAlignment {

 BottomCenter, BottomLeft, BottomRight,

 MiddleCenter, MiddleLeft, MiddleRight,

 TopCenter, TopLeft, TopRight

}

Другим объектом нашего интереса является свойство FlatStyle. Оно используется для управления общим внешним видом элемента управления Button, и ему может быть присвоено любое значение из перечня FlatStyle (определенного в пространстве имен System.Windows.Forms).

public enum System.Windows.Forms.FlatStyle {

 Flat, Popup, Standard, System

}

Для демонстрации возможностей использования типа Button создайте новое приложение Windows Forms с именем Buttons. В окне проектирования добавьте в форму три типа Button (с именами btnFlat, btnPopup и btnStandard) и установите для каждого Button соответствующие значения свойства FlatStyle (FlatStyle.Flat, FlatStyle.Popup и FlatStyle.Standard). Также установите для каждого Button подходящие значения свойства Text и обработайте событие Click для кнопки btnStandard. Как вы вскоре увидите, при щелчке пользователя на этой кнопке позиция текста на кнопке будете меняться в результате изменения значения свойства TextAlign.

Теперь добавьте последний, четвертый элемент Button (с именем btnImage) для поддержки фонового изображения (устанавливаемого с помощью свойства BackgroundImage) и маленькую пиктограмму (устанавливаемую с помощью свойства Image), которая тоже будет динамически перемещаться при щелчке на кнопке btnStandard. Для свойств BackgroundImage и Image вы можете использовать любые файлы изображений, и подходящие файлы изображений есть в папке с исходным программным кодом примера.

Средства проектирования формы создают почти весь необходимый подготовительный программный код в InitializeComponent(), и нам остается только использовать перечень ContentAlignment для перемещения текста на btnStandard и пиктограммы на btnImage. В следующем фрагменте программного кода обратите внимание на то, что для получения списка имен из перечня ContentAlignment вызывается статический метод Enum.GetValues().

partial class MainWindow: Form {

**// Используется для текущего значения выравнивания текста.**

 ContentAlignment currAlignment = ContentAlignment.MiddleCenter;

 int currEnumPos = 0;

 public MainWindow() {

  InitializeComponent();

  CenterToScreen();

 }

 private void btnStandard\_Click (object sender, EventArgs e) {

**// Получение всех значений перечня ContentAlignment,**

  Array values = Enum.GetValues(currAlignment.GetType());

**// Чтение текущей позиции в перечне**

**// и циклический возврат.**

  currEnumPos++;

  if (currEnumPos ›= values.Length) currEnumPos = 0;

**// Чтение текущего значения перечня.**

  currAlignment = (ContentAlignment)Enum.Parse(currAlignment.GetType(), values.GetValue(currEnumPos).ToString());

**// Вывод текста и его выравнивание на btnStandard.**

  btnStandard.TextAlign = currAlignment;

  btnStandard.Text = сurrAlignment.ToString();

**// Размещение пиктограммы на btnImage.**

  btnImage.ImageAlign = currAlignment;

 }

}

Теперь запустите свою программу. При щелчке на средней кнопке вы увидите, что текст займет позицию и изменится в соответствии с текущим значением переменной currAlignment. Пиктограмма в пределах btnImage тоже займет позицию, cоответствующую этому значению. На рис. 21.7 показан вывод программы.

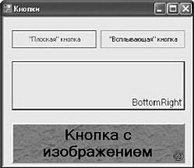


Рис. 21.7. Вариации типа Button

**Исходный код.**Проект Buttons размещён в подкаталоге, соответствующем главе 21.

**Элементы CheckBox, RadioButton и Group Box**

Пространство имен System.Windows.Forms определяет целый ряд других типов, расширяющих возможности ButtonBase, и это, в частности, тип CheckBox (кнопка с независимой фиксацией, может поддерживать до трех возможных состояний) и тип RadioButton (кнопка с зависимой фиксацией, может иметь два состояния – "включена" и "выключено"). Подобно типу Button, эти типы тоже наследуют заметную долю своих функциональных возможностей от базового класса Control. Однако каждый класс определяет и свои уникальные дополнительные возможности. Сначала мы рассмотрим базовые свойства элемента управлений CheckBox, описанные в табл. 21.5.

Таблица 21.5. Свойства CheckBox

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Appearance | Настраивает вид элемента управления Checkbox, используя значения перечня Appearance |
| AutoCheck | Считывает или устанавливает значение, являющееся индикатором необходимости автоматического изменения значений Checked или CheckState и внешнего вида CheckBox при щелчке на нем |
| CheckAlign | Считывает или устанавливает параметры выравнивания по горизонтали и вертикали для CheckBox, используя значения перечня ContentAlignment (во многом аналогично типу Button) |
| Checked | Возвращает булево значение, представляющее состояние CheckBox (включен или выключен). Если свойство ThreeState равно true (истина), то свойство Checked возвращает true как для включенного, так и для неопределенного состояния |
| CheckState | Считывает или устанавливает значение-индикатор включенного состояния CheckBox, используя значения перечня CheckState, а не булево значение |
| ThreeState | Индикатор поддержки в CheckBox не двух, а трех состояний выбора (в соответствии с перечнем CheckState) |

Тип RadioButton не требует пространных комментариев, поскольку этот тип представляет собой лишь немного модифицированный CheckBox. Члены RadioButton почти идентичны членам типа CheckBox. Единственной существенной разницей оказывается поддержка события CheckedChanged, которое (как и следует ожидать) генерируется тогда, когда изменяется значение Checked. Кроме того, тип RadioButton не поддерживает свойство ThreeState, поскольку кнопка типа RadioButton должна быть или включена, или выключена.

Обычно наборы объектов RadioButton группируются вместе, чтобы функционировать, как целое, в соответствии с логикой создаваемой формы. Например, для набора из четырех типов RadioButton. обеспечивающих выбор цвета для автомобиля, обычно нужно гарантировать, чтобы в каждый момент времени мог быть отмечен только один из этих типов. Вместо создания вручную программного кода для решения этой задачи, можно просто использовать элемент управления GroupBox (групповой блок), гарантирующий, что все соответствующие Типы RadioButton будут работать во взаимоисключающем режиме.

Чтобы проиллюстрировать работу с CheckBox, RadioButton и GroupBox, мы создадим новое приложение Windows Forms с именем CarConfig, которое будет расширено в следующих нескольких разделах. Главная форма позволяет пользователю ввести (и подтвердить) информацию о новом транспортном средстве, которое пользователь намеревается купить. Резюме заказа отображается типом Label после щелчка на кнопке Подтвердить заказ. На рис. 21.8 показан исходный вид соответствующего пользовательского интерфейса.

Если для построения формы вы используете окно проектирования формы, вы получите множество членов-переменных, соответствующих каждому элементу графического интерфейса. Соответственно должен будет обновлен метод InitializeComponent().

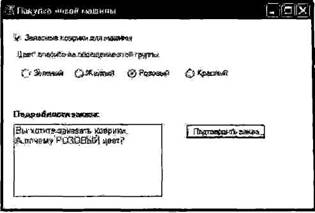


Рис. 21.8 Исходный пользовательский интерфейс формы CarConfig

Первой нашей задачей является настройка типа CheckBox. Как и в случае любого другого производного от Control типа, после установки внешнего вида элемента управления его следует добавить во внутреннюю коллекцию элементов управления формы.

private void InitializeComponent() {

 …

**// checkFloorMats**

**//**

 this.checkFloorMats.Name = "checkFloorMats";

 this.checkFloorMats.Text = "Запасные коврики для машины";

 this.Controls.Add(this.checkFloorMats);

 …

}

Затем нужно сконфигурировать GroupBox и содержащиеся в нем типы RadioButton. Чтобы разместить элемент управления в рамках GroupBox, нужно добавить элемент в коллекцию Controls типа GroupBox (точно так же, как вы добавляли элементы управления в коллекцию Controls формы). Чтобы сделать ситуацию интереснее, используйте окно свойств и задайте обработку событий Enter и Leave для объекта GroupBox, как показано ниже.

private void InitializеComponent() {

 …

**// RadioRed**

 this.radioRed.Name = "radioRed";

 this.radioRed.Size = nеw System.Drawing.Size(04, 23);

 this.radioRed.Text = ''Красный";

**//**

**// groupBoxColor**

**//**

 …

 this.groupBoxColor.Controls.Add(this.radioRed);

 this.groupBoxColor.Text = "Цвет";

 this.groupBoxColor.Enter += new System.EventHandler(this.groupBoxColor\_Enter);

 this.groupBoxColor.Leave += new System.EventHandler(this.groupBoxColor\_Leave);

 …

}

Понятно, что нет никакой необходимости выполнять захват событий Enter и Leave в GroupBox. Однако, для примера, обновите в обработчиках событий текст заголовка GroupBox, как показано ниже.

**// Индикация посещения группы.**

private void groupBoxColor\_Leave(object sender, EventArgs e) {

 groupBoxColor.Text = "Цвет: спасибо, за посещение этой группы…";

}

private void groupBoxColor\_Enter(object sender, EventArgs e) {

 groupBoxColor.Text = "Цвет: вы находитесь в этой группе…";

}

Последними элементами графического интерфейса в этой форме будут типы Label и Button, которые также будут сконфигурированы и вставлены в коллекцию Controls формы с помощью InitializeComponent(). Тип Label используется для отображения информации заказа, формирующейся в обработчике события Click кнопки Button подтверждения заказа, как показано ниже.

private void btnOrder\_Click(object sender, System.EventArgs e) {

**// Построение строки для отображения информации.**

 string orderInfo = "";

 if (checkFloorMats.Checked) orderInfo += "Вы хотите заказать коврики.\n";

 if (radioRed.Checked) orderInfo += "Вы выбрали красный цвет.\n";

 if (radioYellow.Checked) orderInfo += "Вы выбрали желтый цвет.\n";

 if (radioGreen.Checked) orderInfo += "Вы выбрали зеленый цвет.\n";

 if (radioPink.Checked) orderInfo += "А почему РОЗОВЫЙ цвет?\n";

**// Отправка строки элементу Label.**

 infoLabel.Text = orderInfo;

}

Обратите внимание на то, что как CheckBox, так и RadioButton поддерживают свойство Checked, которое позволяет выяснить текущее состояние элемента. Кроме того, напомним, что если вы сконфигурировали CheckBox с тремя состояниями, то состояние элемента нужно проверять с помощью свойства CheckState.

**Элемент CheckedListBox**

Теперь, завершив исследование базовых элементов управления Button, давайте рассмотрим набор типов списка, в частности CheckedListBox, ListBox и ComboBox. Элемент управления CheckedListBox (окно отмечаемого списка) позволяет сгруппировать соответствующие элементы CheckBox в список, допускающий прокрутку. Предположим, что вы добавили в форму элемент управления CarConfig, дающий пользователю возможность указать на выбор ряд характеристик, которым должна удовлетворять система звуковоспроизведения автомобиля (рис. 21.9).

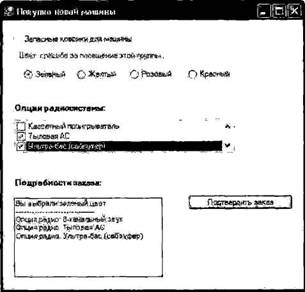


Рис. 21.9. Тип CheckedListBox

Чтобы добавить в CheckedListBox новые элементы, вызовите Add() для каждого элемента или используйте метод AddRange() с массивом объектов (строк, если быть точным), представляющих весь набор отмечаемых элементов управления. Следует знать о том, что в режиме проектирования любой тип списка можно заполнить с помощью свойств Items в окне свойств (просто щелкните на кнопке с многоточием и введите подходящие строковые значения). Вот часть программного кода InitializeComponent(), соответствующая конфигурации CheckedListBox.

private void InitializeComponent() {

 …

**// checkedBoxRadioOptions**

**//**

 this.checkedBoxRadioOptions.Items.AddRange(new object[] {

  "Фронтальная АС", "8-канальный звук",

  "CD-проигрыватель", "Кассетный проигрыватель",

  "Тыловая AC", "Ультра-бас(сабвуфер)"

 });

 …

 this.Controls.Add(this.checkedBoxRadioOptions);

}

Теперь обновите логику обработки события Click для кнопки Подтвердить заказ. Выясните, какие из элементов CheckedListBox в настоящий момент отмечены, и добавьте их в строку orderInfo. Вот как должен выглядеть соответствующий программный код.

private void btnOrder\_Click(object sender, EventArgs e) {

**// Построение строки *с*информацией для отображения.**

 string orderInfo = "";

 …

 orderInfo += "-------------------------------\n";

**// Для каждого элемента из CheckedListBox.**

 for (int i = 0; i ‹ checkedBoxRadioOptions.Items.Count; i++) {

**// Отмечен ли элемент?**

  if (checkedBoxRadioOptions.GetItemChecked(i)) {

**// Получение текста элемента и добавление к orderInfo.**

   orderInfo += "Опция радио: ";

   orderInfo += checkedBoxRadioOptions.Items[i].ToString();

   orderInfo += "\n";

  }

 }

 …

}

В качестве заключительного замечания относительно типа CheckedListBox обращаем ваше внимание на то, что этот тип поддерживает многоколоночное представление, устанавливаемое с помощью унаследованного свойства MultiColumn. Поэтому, если вы добавите в программный код оператор

checkedBoxRadioOptions.MultiColumn = true;

вы увидите многоколоночный CheckedListBox, как показано на рис. 21.10.

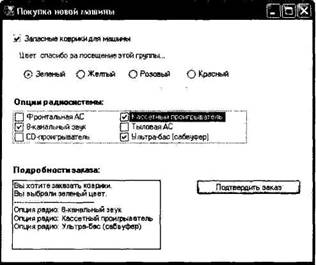


Рис. 21.10. Многоколоночный тип CheckedListBox

**Элемент Listbox**

Как уже упоминалось выше, тип CheckedListBox наследует большинство своих возможностей от типа ListBox (окно списка). Чтобы продемонстрировать возможности использования типа ListBox, давайте добавим в наше приложение CarConfig возможность выбора пользователем марки автомобиля (BMW, Yugo и т.д.). Нa рис. 21.11 показан внешний вид того пользовательского интерфейса, который мы хотим получить.

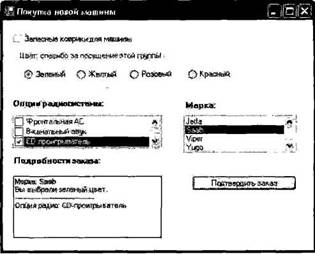


Рис. 21.11. Тип ListBox

Как всегда, начните с создания члена-переменной для работы с типом (в данном случае это тип ListBox). Затем сконфигурируйте элемент управления в соответствии со следующим фрагментом из InitializeComponent().

private void InitializeComponent() {

 …

**// carMakeList**

**//**

 this.carMakeList.Items.AddRange(new object[] {"BMW", "Caravan", "Ford", "Grand Am", "Jeep", "Jetta", "Saab", "Viper", "Yugo"});

 …

 this.Controls.Add(this.carMakeList);

}

Изменения обработчика событий btnOrder\_Click() также очень просты.

private void btnOrder\_Click(object sender, EventArgs e) {

**// Построение строки для отображения информации.**

 string orderInfo = "";

 …

**// Получение выбранного элемента (не индекса!)**.

 if (carMakeList.SelectedItem != null) orderInfo += "Марка: " + carMakeList.SelectedItem + "\n";

 …

}

**Элемент ComboBox**

Подобно ListBox, тип ComboBox (комбинированное окно) позволяет пользователю сделать выбор из вполне определенного набора возможностей. Однако тип ComboBox уникален в том, что пользователю также позволяется вставить дополнительные элементы. Напомним, что ComboBox получается из ListBox (а последний, в свою очередь, получается из Control). Для иллюстрации возможностей использования рассматриваемого элемента добавьте в форму приложения CarConfig еще один элемент управления, который позволит ввести имя продавца, с которым пользователь предпочитает иметь дело. Если имени нужного продавца в списке нет, пользователь может ввести соответствующее имя. Одна из возможных модификаций интерфейса показана на рис 21.12 (можете назначить продавцам такие имена, какие захотите).

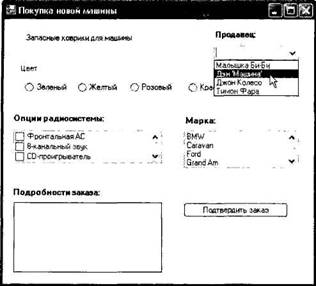


Рис. 21.12. Тип ComboBox

Соответствующая модификация начинается с настройки самого ComboBox. Как видите, используемая здесь программная логика аналогична логике ListBox.

private void InitializeComponent() {

 …

**// comboSalesPerson**

**//**

 this.comboSalesPerson.Items.AddRange(new object[] {"Малышка Би-Би", "Дэн \' Машина\'", "Джой Колесо", "Тимон Фара"});

 …

 this.Controls.Add(this.comboSalesPerson);

}

Модификация обработчика событий btnOrder\_Click() снова оказывается очень простой.

private void btnOrder\_Click(object sender, EventArgs e) {

**// Построение строки для отображения информации.**

 string orderInfo = "";

**// Использование свойства Text для имени продавца,**

**// указанного пользователем.**

 if (comboSalesPerson.Text != "") orderInfo += "Продавец: " + comboSalesPerson.Text + "\n";

 else orderInfo *+=*"Вы не указали имя продавца!" + "\n";

 …

}

**Порядок переходов по нажатию клавиши табуляции**

Теперь, когда вы создали достаточно интересную форму, давайте рассмотрим проблему порядка переходов по нажатию клавиши табуляции. Вы, наверное, знаете, что в том случае, когда форма содержит множество элементов графического интерфейса, пользователь может переместить фокус ввода от одного элемента к другому, используя клавишу табуляции. Настройка порядка переходов по табуляции для набора элементов управления требует понимания сути двух ключевых свойств: TabStop и TabIndex.

Свойству TabStop можно присвоить значение true (истина) или false (ложь), в зависимости от того, хотите вы или нет, чтобы соответствующий элемент графического интерфейса был доступен по нажатию клавиши табуляции. Если свойству TabStop данного элемента управления присвоено true, то свойству TabOrder устанавливается значение (начиная с нуля), соответствующее порядку активизации этого элемента управления в последовательности нажатий клавиш табуляции. Рассмотрите следующий пример.

**// Настройка свойств табуляции.**

radioRed.**TabIndex**= 2;

radioRed.**TabStop**= true;

**Мастер настройки переходов по табуляции**

В Visual Studio 2005 IDE есть мастер настройки переходов по табуляции, доступ к которому можно получить с помощью выбора View→Tab Order из меню (этот пункт меню доступен только при активном окне проектирования формы). После активизации мастера ваша форма в режиме проектирования будет отображать текущие значения TabIndex всех элементов, Чтобы изменить эти значения, щелкните на порядковом номере выбранного вами элемента (рис. 21.13). Чтобы выйти из мастера настройки переходов по табуляции, достаточно нажать ‹Esc›.



Рис. 21.13. Мастер настройки переходов по табуляции

**Установка кнопки, выбираемой по умолчанию**

Многие формы, предназначенные для пользовательского ввода (особенно диалоговые окна), предполагают наличие кнопки, которая автоматически отвечает на нажатие пользователем клавиши ‹Enter›. Если вы хотите, чтобы при нажатии пользователем клавиши ‹Enter› в нашей форме автоматически вызывался обработчик события Click для btnOrder, просто установите свойство AcceptButton формы так, как показано ниже.

**// При нажатии ‹Enter› все будет происходить так, как будто**

**// пользователь щелкнул на кнопке btnOrder.**

this**.AcceptButton**= btnOrder;

**Замечание.**В форме можно также имитировать щелчок на кнопке Cancel (Отмена) при нажатии пользователем клавиши ‹Esc›. Для этого нужно назначить свойству CancelButton имя объекта Button, представляющего кнопку Cancel (Отмена).

**Работа с другими элементами управления**

Итак, мы с вами выяснили, как работать большинством базовых элементов управления Windows Forms (Label, TextBox, и т.д.). Следующей задачей будет рассмотрение элементов графического интерфейса, обладающих более сложными функциональными возможностями. К счастью, только то, что элемент управления выглядит "более экзотическим", обычно означает не то, что с таким элементом будет трудно работать, а то, что вам потребуется немного больше времени на его освоение. На следующих нескольких страницах мы рассмотрим следующие элементы графического интерфейса.

• MonthCalendar

• ToolTip

• TabControl

• TrackBar

• Panel

• Элементы управления UpDown

• ErrorProvider

• TreeView

• WebBrowser

Для начала давайте завершим проект CarConfig, рассмотрев элементы управления MonthCalendar и ToolTip.

**Элемент MonthCalendar**

Пространство имен System.Windows.Forms предлагает очень полезный элемент управления MonthCalendar, который дает пользователю возможность выбрать дату (или диапазон дат), используя дружественный интерфейс. Чтобы продемонстрировать этот элемент управления, обновим приложение CarConfig так, чтобы пользователь мог ввести дату доставки купленного транспортного средства. На рис. 21.14 показана обновленная (и слегка модифицированная в отношении размещения элементов) форма.

Элемент управления MonthCalendar имеет весьма широкие возможности, и с его помощью очень просто программно выполнить "захват" диапазона дат, выбранных пользователем. Поведением по умолчанию для этого типа является автоматический выбор (и выделение) текущей (сегодняшней) даты.

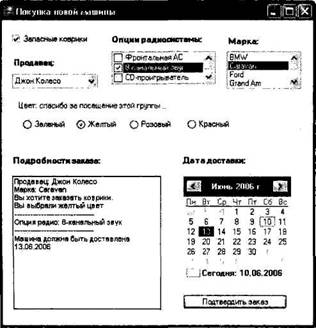


Рис. 21.14. Тип MonthCalendar

Чтобы получить выбранную в настоящий момент дату программно, можно обновить обработчик события Click для Button, как предлагается ниже.

private void btnOrder\_Click(object sender, EventArgs e) {

**// Построение строки для отображения информации.**

 string orderInfo = "";

 …

**// Получение даты доставки.**

 DateTime d = monthCalendar.SelectionStart;

 string dateStr = string.Format("{0}.{1}.{2}", d.Day, d.Month, d.Year);

 orderInfo += "Машина должна быть доставлена\n" + dateStr;

 …

}

Обратите внимание на то, что информацию о выбранной в настоящий момент дате у элемента управления MonthCalendar можно запросить с помощью свойства SelectionStart. Это свойство возвращает ссылку DateTime, которая запоминается в локальной переменной. Используя свойства типа DateTime, можно извлечь нужную информацию в подходящем вам пользовательском формате.

В настоящий момент мы предполагаем, что пользователь указал только один день для доставки нового автомобиля. Но что делать, если вы хотите предоставить пользователю возможность выбора целого диапазона приемлемой даты доставки? В этом случае пользователю нужно просто "Протащить" указатель мыши через соответcтвующий диапазон дат. Вы уже видели, как можно получить начало диапазона выделения, используя свойство SelectionStart. Конец диапазона выделения можно определить с помощью свойства SelectiоnEnd. Вот как должен выглядеть программный код, модифицированный соответствующим образом.

private void btnOrder\_Click(object sender, EventArgs e) {

**// Построение строки для отображения информации.**

 string orderInfo = "";

 …

**// Получение диапазона дат доставки**…

 DateTime startD = monthCalendar.**SelectionStart**;

 DateTime endD *=*monthCalerdar.**SelectionEnd**;

 string.dateStartStr = string.Format("{0}.{1}.{2}", stаrtD.Day, startD.Month, startD.Year);

 string dabeEndStr = string.Format("{0}.{1}.{2}", endD.Daу, endD.Month, endD.Year);

**// Тип DateTime поддерживает перегруженные операции!**

 if (dateStartStr != dateEndStr) {

  orderInfo += "Машина должна быть доставлена\n" + "c " + dateStartStr + "по " + dateEndStr;

 } else **// Когда выбрано одно значение даты.**

  orderInfo += "Машина должна быть доставлена\n" + dateStartStr;

 …

}

Замечание. Windows Forms содержит элемент управления DateTimePicker, который позволяет предложить MonthCalendar в раскрывающемся элементе управления DropDown.

**Элемент ToolTip**

В связи с рассматриваемой формой CarConfig мы должны продемонстрировать еще одну, заключительную, возможность. Многие современные интерфейсы пользователя предлагают так называемые*всплывающие подсказки.*В пространстве имен System.Windows.Forms эта возможность представлена типом ToolTip. Соответствующие подсказки представляют собой небольшие плавающие окна, в которых отображаются вспомогательные сообщения, когда курсор задерживается вблизи данного элемента интерфейса.

Для примера добавьте такую подсказку для календаря CarConfig. Сначала перетащите новый элемент управления ToolTip из панели инструментов в окно проектирования формы и переименуйте этот элемент управления в calendarTip. Внешний вид элемента ToolTip можно задать, используя окно свойств, например.

private void InitializeComponent () {

 …

**// calendarTip**

**//**

 this.calendarTip.isBalloon = true;

 this.calendarTip.ShowAlways = true;

 this.calendarTip.ToolTipIcon = System.Windows.Forms.ToolTipIcon.Info;

 …

}

Чтобы связать ToolTip с данным элементом управления, выберите элемент управления, в котором должен активизироваться ToolTip, и соответствующим образом установите свойство ToolTip on (рис. 21.15).

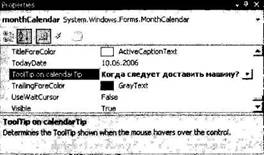


Рис. 21.15. Ассоциация ToolTip с элементом управления

Теперь наш проект CarConfig можно считать завершенным. На рис. 21.16 созданная подсказка показана в действии.



Рис. 21.16. Элемент ToolTip в действии

**Исходный код**. Проект CarConfig размещен в подкаталоге, соответствующем главе 21.

**Элемент TabControl**

Чтобы проиллюстрировать остальные "экзотические" элементы управления, давайте построим новую форму, поддерживающую TabControl (элемент управления вкладками). Вы, возможно, знаете, что TabControl позволяет селективно скрывать или показывать страницы связанного содержимого с помощью щелчка на соответствующих "закладках". Сначала создайте новое приложение Windows Forms с именем ExoticControls и поменяйте имя исходной формы на MainWindow.

Затем добавьте TabControl в окно проектирования формы и, используя окно свойств, с помощью коллекции Tab Pages откройте редактор страниц (в соответствующей строке окна свойств щелкните на кнопке с многоточием). Появится диалоговое окно конфигурации инструмента. Добавьте в нем шесть страниц и установите свойства Text и Name страниц в соответствии с тем, как показано на рис. 21.17.



Рис. 21.17. Многостраничный элемент управления TabControl

При создании элемента управления TabControl следует учитывать то, что каждая страница представляется объектом TabPage, содержащимся во внутренней коллекции страниц TabControl. Сконфигурированный объект TabControl (подобно любому другому элементу графического интерфейса в форме) добавляется в коллекцию Controls формы. Рассмотрите соответствующий фрагмент метода InitializeComponent().

private void InitializeComponent() {

 …

**// tabControlExoticControls**

**//**

 this.tabControlExoticControls.Controls.Add(this.pageTrackBars);

 this.tabControlExoticControls.Controls.Add(this.pagePanels);

 this.tabControlExoticControls.Controls.Add(this.pageUpDown);

 this.tabControlExoticControls.Controls.Add(this.pageErrorProvider);

 this.tabControlExpticControls.Controls.Add(this.pageTreeView);

 this.tabControlExoticControls.Controls.Add(this.pageWebBrowser);

 this.tabControlExoticControls.Location = new System.Drawing.Point(13, 13);

 this.tabControlExoticControls.Name = "tabControlExoticControls";

 this.tabControlExoticControls.SelectedIndex = 0;

 this.tabControlExoticControls.Size = new System.Drawing.Size(463, 274);

 this.tabControlExoticControls.TabIndex = 0;

 this.Controls.Add(this.tabControlExoticControls);

}

Теперь, имея базовую форму, поддерживающую набор вкладок, вы можете настроить каждую страницу для демонстрации остальных "экзотических" элементов управления. Сначала давайте рассмотрим роль TrackBar.

**Замечание**. Элемент управления TabControl обеспечивает поддержку событий Selected, Selecting, Deselected и Deselecting. Это может оказаться полезным тогда, когда требуется динамически генерировать соответствующие элементы в пределах страницы.

**Элемент TrackBar**

Элемент управления TrackBar дает пользователям возможность выбора из диапазона значений, используя нечто, похожее на полосу прокрутки. При работе с этим типом нужно установить минимальное и максимальное значения диапазона, минимальное и максимальное значения приращения, а также начальное положение ползунка. Соответствующие значения можно установить с помощью свойств, описанных в табл. 21.6.

**Таблица 21.6.**Свойства TrackBar

|  |  |
| --- | --- |
| **Свойства** | **Описание** |
| LargeChange | Число делений, на которое изменяется положение ползунка TrackBar, когда происходит событие, предполагающее "большое" изменение (например, щелчок кнопки мыши, когда указатель находится в области направляющей ползунка, или нажатие клавиш ‹PageUp› и ‹PageDown›) |
| Maximum Minimum | Верхняя и нижняя границы диапазона TrackBar |
| Orientation | Ориентация для TrackBar. Действительными являются значения из перечня Orientation (горизонтальная или вертикальная ориентация) |
| SmallChange | Число делений, на которое изменяется положение TrackBar, когда происходит событие, предполагающее "малое" изменение (например, нажатие клавиш со стрелками) |
| TickFrequency | Влияет на число делений, которое требуется изобразить. Например, для TrackBar c верхним пределом 100 нерационально изображать все 100 делений для элемента управления длиной 5 см. Если установить свойство TickFrequency равным 5, для TrackBar будет показано только 20 делений (одно деление будет представлять 5 единиц) |
| TickStyle | Задает внешний вид элемента управления TrackBar. От этого значения (которое должно соответствовать значениям перечня TickStyle) зависит и то, где будут изображены деления относительно ползунка, и то, как будет выглядеть сам ползунок |
| Value | Читает или устанавливает значение, задающее текущее положение ползунка TrackBar. С помощью этого свойства можно получить числовое значение, содержащееся в TrackBar, чтобы использовать его в приложении |

Для примера обновите первую вкладку элемента TabControl, разместив на ней три элемента TrackBar, для каждого из которых верхнее значение диапазона равно 255, а нижнее – нулю, При смещении пользователем любого из ползунков приложение перехватывает событие Scroll и динамически создает новый тип System.Drawing.Color на основе новых значений ползунков. Этот тип Color будет использоваться для того, чтобы отображать соответствующим цветом элемент PictureBox (с именем colorBox) и соответствующие RGB-значения в пределах типа Label (с именем lblCurrColor). На рис. 21.18 первая страница окна показана в завершенном виде.

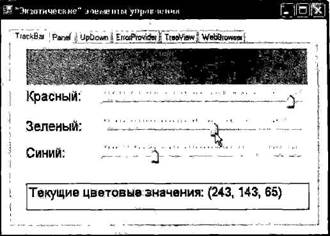


Рис. 21.18. Страница TrackBar

Сначала, используя окно проектирования формы, разместите три элемента управления TrackBar на первой вкладке и назначьте соответствующим членам-переменным подходящие имена (redTrackBar, greenTrackBar и blueTrackBar). Затем обработайте событие Scroll для каждого TrackBar. Вот подходящий программный код InitializeComponent() для blueTrackBar (программный код остальных полос почти идентичен данному, за исключением имени обработчика события Scroll).

private void InitializeComponent() {

 …

**//**

**// blueTrackBar**

**//**

 this.blueTrackBar.Maximum = 255;

 this.blueTrackBar.Name = "blueTrackBar";

 this.blueTrackBar.TickFrequency = 5;

 this.blueTRackBar.TickStyle = System.Windows.Forms.TickStуle.TopLeft;

 this.blueTrackBar.Scroll += new System.EventHandler(this.blueTrackBar.Scroll);

 …

}

Заметим, что минимальным значением по умолчанию для TrackBar является 0, поэтому его явно устанавливать не нужно. В обработчиках событий Scroll для каждого TrackBar выполняется вызов вспомогательной функции UpdateColor(), которую нам еще предстоит написать.

private void blueTrackBar\_Scroll(object sender, EventArgs e) {

 UpdateColor();

}

Функция UpdateColor() отвечает за решение двух главных задач. Во-первых, нужно прочитать текущее значение каждого TrackBar и использовать эти данные для вычисления нового Color с помощью Color.FromArgb(). Имея новый готовый цвет, следует соответствующим образом обновить член-переменную PictureBox (с именем colorBox), чтобы установить текущий цвет фона. Наконец, UpdateColor() комбинирует значения ползунков в строке, размещаемой в элементе Label(lblCurrColor), как показано ниже.

private void UpdateColor() {

**// Получение нового цвета на основе значений ползунков.**

 Color с = Color.FromArgb(redTrackBar.Value, greenTrackBar.Value, blueTrackBar.Value);

**// Изменение цвета в PictureBox.**

 colorBox.BackColor = c;

**// Установка текста для надписи.**

 lblCurrColor.Text = string.Format("Текущие цветовые значения: ({0}, {1}, (2})", redTrackBar.Value, greenTrackBar.Value, blueTrackBar.Value);

}

Заключительным штрихом является установка начальных значений каждого ползунка при начальном появлении формы и отображение текущего цвета, как показано ниже.

public MainWindow() {

 InitializeComponent();

 CenterToScreen();

**// Установка исходного положения ползунков.**

 redTrackBar.Value = 100;

 greenTrackBar.Value = 255;

 blueTrackBar.Value = 0;

 UpdateColor();

}

**Элемент Panel**

Как вы уже видели, элемент управления GroupBox может использоваться для того, чтобы логически объединить ряд элементов управления (например, переключателей) и заставить их функционировать во взаимосвязи. Элемент управления Panel в этом смысля является близким к GroupBox. Элементы управления Panel тоже используются для группировки родственных элементов управления в логические единицы. Одним из различий является то, что тип Panel получается из класса ScrollableControl, поэтому Panel может поддерживать полосы прокрутки, чего нет у GroupBox.

Элементы управления Panel могут также использоваться для "консервации" содержимого экрана. Например, если у вас есть группа элементов управления, которые занимают всю нижнюю половину формы, вы можете поместить эту группу в Panel половинного размера и установить значение true (истина) для свойства AutoScroll. Тогда пользователь сможет использовать полосу (или полосы) прокрутки, чтобы просмотреть весь набор элементов. К тому же, если для свойства BorderStyle элемента Panel установить значение None, то этот тип можно будет использовать для группировки набора элементов, которые очень легко показать или скрыть способом. совершенно прозрачным в отношении конечного пользователя.

Для примера давайте добавим на вторую страницу TabControl два типа Button (с именами btnShowPanel и btnHidePanel) и один тип Panel, который содержит пару текстовых блоков (txtNormalText и txtUpperText) с инструктирующим элементом Label. (Какие именно элементы находятся в Panel, для этого примера не очень важно.) На рис. 21.19 показан окончательный вид соответствующей страницы.

С помощью окна свойств обработайте событие TextChanged для первого элемента TextBox, и в сгенерированном обработчике события поместите в txtUpperText преобразованный в верхний регистр текст, введенный в txtNormalText.

private void txtNormal'Text\_TextChanged(object sender, EventArgs e) {

 txtUpperText.Text = txtNormalText.Text.ToUpper();

}

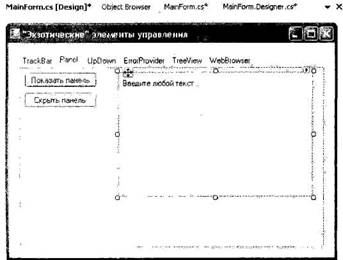


Рис. 21.19. Страница Panel

Теперь обработайте событие Click для каждой кнопки. Как вы можете догадаться, нужно просто скрыть или показать Panel (вместе со всеми содержащимися там элементами пользовательского интерфейса).

private void btnShowPanel\_Click(object sender, EventArgs e) {

 panelTextBoxes.Visible = true;

}

private void btnHidePanel\_Click(object sender, EventArgs e) {

 panelTextBoxes.Visible = false;

}

Если теперь выполнить программу и щелкнуть на той или другой кнопке в соответствующем окне, вы обнаружите, что содержимое Panel соответственно показывается и скрывается. Конечно, этот пример не производит слишком большого впечатления, но я уверен, что вы смогли увидеть его возможности. Например, вы можете иметь пункт меню или окно безопасности, способные предоставить пользователю "простой" или "сложный" набор элементов. Вместо того чтобы вручную устанавливать свойство Visible равным false (ложь) для множества элементов, вы можете группировать их в пределах Panel и соответственно установить одно свойство Visible.

**Элементы UpDown**

В рамках Windows Forms предлагается два элемента, функционирующие, как *элементы управления с прокруткой*(также известные, как *элементы управления UpDown).*Подобно ComboBox и ListBox, эти новые элементы также позволяют пользователю выбрать элемент из некоторого диапазона возможных элементов.

Разница в том, что при использовании элемента управления DomainUpDown или NumericUpDown варианты выбираются с помощью небольших стрелок, направляющих вверх и вниз. Взгляните, например, на рис. 21.20.

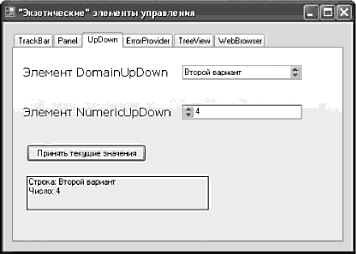


Рис. 21.20. Работа с типами UpDown

С учетом того, что вы уже освоили работу с подобными типами, вы не должны встретить особых сложностей при работе с элементами UpDown. Элемент DomainUpDown дает пользователю возможность сделать выбор из набора строковых данных. Элемент NumericUpDown позволяет выбрать подходящие значений из диапазона числовых данных. Каждый из этих элементов является прямым потомком общего базового класса UpDownBase. В табл. 21.7 описаны некоторые важные свойства этого класса.

**Таблица 21.7.**Свойства UpDownBase

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| InterceptArrowKeys | Читает или устанавливает значение, являющееся индикатором того, что пользователю разрешено использовать стрелки вверх и вниз для выбора значений |
| ReadOnly | Читает или устанавливает значение, являющееся индикатором того, что текст разрешается менять только с помощью стрелок вверх и вниз, но не с помощью ввода в элемент управления с клавиатуры с целью поиска данной строки |
| Text | Читает или устанавливает текущий текст, отображаемый в элементе управления с прокруткой |
| TextAlign | Читает или устанавливает значение, задающее параметры выравнивания текста в элементе управления с прокруткой |
| UpDownAlign | Читает или устанавливает значение, задающее параметры выравнивания стрелок вверх и вниз в элементе управления с прокруткой, в соответствии со значениями перечня LeftRightAlignment |

Элемент управления DomainUpDown добавляет небольшой набор свойств, позволяющих конфигурировать и обрабатывать текстовые данные этого элемента (табл. 21.8).

**Таблица 21.8.**Свойства DomainUpDown

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Items | Позволяет получить доступ к множеству элементов, хранимых в данном элементе управления |
| SelectedIndex | Возвращает индекс выбранного в настоящий момент элемента (отсчет начинается с нуля, значение -1 указывает отсутствие выбора) |
| SelectedItem | Возвращает выбранный элемент (а не его индекс) |
| Sorted | Индикатор необходимости упорядочения строк по алфавиту |
| Wrap | Индикатор необходимости циклического возвращения к первому или последнему элементу, когда пользователь достигает крайних элементов списка |

Элемент NumericUpDown так же прост (табл. 21.9).

**Таблица 21.9.**Свойства NumericUpDown

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| DecimalPlaces ThousandsSeparator Hexadecimal | Используются для указания правил отображения числовых данных |
| Increment | Устанавливает числовое значение приращения для элемента управления при щелчке на стрелке вверх или вниз. Значением по умолчанию для приращения является 1 |
| Minimum Maximum | Устанавливают верхнюю и нижнюю границы значений для элемента управления |
| Value | Возвращает текущее значение элемента управления |

Вот та часть InitializeComponent (), которая задает конфигурацию NumericUpDown и DomainUpDown на этой странице.

private void InitializeComponent() {

 …

**//**

**// numericUpDown**

**//**

 …

 this.numericUpDown.Maximum = new decimal(new int[] {5000, 0, 0, 0});

 this.numericUpDown.Name = "numericUpDown";

 this.numericUpDown.Thousands.Separator = true;

**//**

**// domainUpDown**

**//**

 this.domainUpDown.Items.Add("Второй вариант");

 this.domainUpDown.Items.Add("Последний вариант");

 this.domainUpDown.Items.Add("Первый вариант");

 this.domainUpDown.Items.Add("Третий вариант");

 this.domainUpDown.Name = "domainUpDown";

 this.domainUpDown.Sorted = true;

 …

}

Обработчик события Click для типа Button этой страницы просто запрашивает у каждого типа его текущее значение и размещает его в рамках подходящего типа Label (с именем lblCurrSel) в виде форматированной строки, как показано ниже.

private void ntnGetSelections\_Click(object sender, EventArgs e) {

**// Получение информации от элементов UpDown.…**

 lblCurrSel.Text = string.Format("Строка: {0}\nЧисло: {1}", domainUpDown.Text, numericUpDown.Value);

}

**Элемент ErrorProvider**

В большинстве приложений Windows Forms приходится, так или иначе, проверять правильность пользовательского ввода. Это особенно касается диалоговых окон, поскольку вы должны информировать пользователя о том, что он сделал ошибку, прежде чем пользователь продолжит ввод. Тип ErrorProvider может использоваться для того, чтобы обеспечить пользователю визуальные подсказки в отношении ошибок ввода. Предположим, например, что у вас есть форма, содержащая элементы TextBox и Button. Если пользователь введет в TextBox более пяти символов и TextBox утрачивает фокус ввода, можно отобразить информацию, показанную на рис. 21.21.

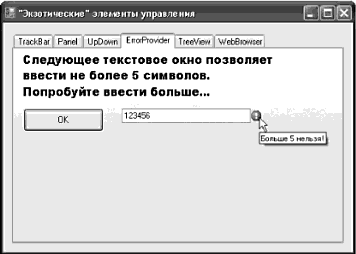


Рис. 21.21 Действие ErrorProvider

Здесь вы обнаруживаете, что пользователь ввел более пяти символов, и в ответ размещаете небольшую пиктограмму ошибки (!) рядом с объектом TextBox. Если пользователь подведет указатель мыши к этой пиктограмме, появится "всплывающий" текст с описанием ошибки. Кроме того, этот элемент ErrorProvider сконфигурирован так, чтобы заставить пиктограмму "мигать", что усилит визуальное воздействие (конечно, без запуска приложения вы этого не увидите).

Если вы хотите использовать такой вид проверки ввода, то вам. прежде всего, нужно изучить свойства класса Control, описания которых приводятся в табл. 21.10.

**Таблица 21.10.**Свойства и события Control

|  |  |
| --- | --- |
| **Свойство или событие Описание** |  |
| CausesValidation | Индикатор того, что выбор этого элемента управления вызывает проверку ввода для элементов управления, требующих такой проверки |
| Validated | Событие, генерируемое тогда, когда элемент управления заканчивает выполнение программной логики проверки ввода |
| Validating | Событие, генерируемое тогда, когда элемент управления проверяет пользовательский ввод (например, когда элемент управления утрачивает фокус ввода) |

Каждый элемент графического интерфейса может установить для свойства CausesValidation значение true (истина) или false (ложь), причем значением по умолчанию является true. Если вы установите для указанных данных значение true, данный элемент управления при получении им фокуса ввода заставит остальные элемент управления в форме выполнить проверку ввода. При получении фокуса ввода проверяющим элементом управления генерируются события Validating и Validated для каждого элемента управления. В контексте обработчика события Validating вы должны конфигурировать соответствующий ErrorProvider. Также можно, но необязательно, обработать событие Validated, чтобы определить, когда элемент управления закончит цикл проверки.

Тип ErrorProvider предлагает очень небольшой набор членов. Для нашего примера самым важным является свойство BlinkStyle, которому можно присвоить любое значение из перечня ErrorBlinkStyle. Описания этих значений даются в табл. 21.11.

**Таблица 21.11.**Значения ErrorBlinkStyle

|  |  |
| --- | --- |
| **Значение** | **Описание** |
| AlwaysBlink | Заставляет пиктограмму ошибки "мигать", когда ошибка отображается впервые или когда элементу управления назначается строка новой ошибки, а пиктограмма ошибки уже отображается |
| BlinkIfDifferentError | Заставляет пиктограмму ошибки "мигать", когда пиктограмма ошибки уже отображается, но элементу управления назначается строка новой ошибки |
| NeverBlink | Индикатор того, что пиктограмма ошибки не должна "мигать" никогда |

Для примера добавьте на вкладку ErrorProvider элементы управления Button, TextBox и Label, как показано на рис. 21.21. Затем перетащите в окно проектирования формы элемент ErrorProvider и присвойте этому элементу имя tooManyCharactersErrorProvider. Вот соответствующий фрагмент программного кода InitializeComponent().

private void InitializeComponent() {

 …

**//**

**// tooManyCharactersErrorProvider**

**//**

 this.tooManyCharaсtersErrorProvider.BlinkRate = 500;

 this.tooManyCharactersErrorProvider.BlinkStyle*=*System.Windows.Forms.ErrorBlinkStyle.AlwaysBlink;

 this.tooManyCharactersErrorProvider.ContainerControl = this;

}

После настройки внешнего вида ErrorProvider вы должны выполнить привязку ошибки к TextBox в контексте обработчика события Validating, как показано ниже.

private void txtInput\_Validating(object sender, CancelEventArgs е) {

**// Длина текста меньше 5?**

 if (txtInput.Text.Length › 5) {

  errorProvider1.SetError(txtInput, "Больше 5 нельзя!");

 } else **// Все в порядке, не показывать ничего.**

  errorProvider1.SetError(txtInput, ");

}

**Элемент TreeView**

Элементы управления TreeView очень полезны тем, что они позволяют визуально отображать иерархии данных (например, структуру каталогов или любую другую структуру, связанную отношением "родитель-потомок"). Элемент управления TreeView предлагает очень широкие возможности настройки. При желании вы можете добавить пользовательские изображения, задать цвет узлов, элементы контроля узла и другие визуальные усовершенствования. (Заинтересованным читателям за дополнительной информацией об этом элементе управления предлагается обратиться к документации .NET Framework 2.0 SDK.)

Чтобы продемонстрировать основные возможности использования TreeView, на следующей странице вашего TabControl мы программно разместим элемент TreeView, определяющий ряд узлов наивысшего уровня, представляющих набор типов Car (автомобиль), Каждый узел Car имеет два подчиненных узла, представляющих текущую скорость автомобиля и любимую радиостанцию водителя. На рис. 21.22 обратите внимание на то, что выбранный элемент выделен подсветкой. Также заметьте, что в области элемента Label кроме выбранного узла отображаются имена родительского и следующего узлов (если последние имеются).

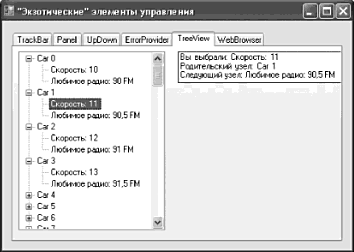


Рис. 21.22. Элемент TreeView в действии

Предполагая, что соответствующий пользовательский интерфейс скомпонован из элементов управления TreeView (с именем treeViewCars) и Label (с именем lblNodeInfo), добавьте в свой проект ExoticControls новый файл C#, который моделирует тривиальный типа Car, имеющий Radio.

namespace ExoticControls {

 class Car {

  public Car(string pn, int cs) {

   petName = pn;

   currSp = cs;

  }

  public string petName;

  public int currSp;

  public Radio r;

 }

 class Radio {

  public double favoriteStation;

  public Radio(double station) { favoriteStation = station; }

 }

}

Производный от Form тип будет поддерживать обобщенный тип List‹› (с именем listCars), содержащий 100 типов Car, которые будут занесены в список в конструкторе типа MainForm, заданном по умолчанию. Кроме того, этот конструктор вызывает новый вспомогательный метод BuildCarTreeView(), который не имеет никаких аргументов и возвращает void. Вот соответствующая модификация программного кода.

public partial class MainWindow: Form {

**// Создание нового List для хранения объектов Car.**

 private List‹Car› listCars = new List‹Car›();

 public MainWindow() {

  …

**// Заполнение ячеек List‹› и построение TreeView.**

  double offset = 0.5;

  for (int x = 0; x ‹ 100; x++) {

   listCars.Add(new Car(string.Format("Car {0}", x)*,*10 + x));

   offset += 0.5;

   listCars[x].r = new Radio(89.0 + offset);

  }

  BuildCarTreeView();

 }

 …

}

Обратите внимание на то, что petName каждого автомобиля задается на основе текущего значений x (Car 0, Car 1, Car 2 и т.д.). Текущая скорость образуется путем сдвига x на 10 (от 10 км/ч до 109 км/ч), а любимая радиостанция задается сдвигом от начального значения 89.0 на 0.5 (90, 90.5, 91, 91.5 и т.д.).

Итак, у вас есть список Car, и вам нужно спроецировать эти значения на узлы элемента управления TreeView. Здесь самое важное – понять, что каждый узел, как высшего уровня, так и подчиненного, представляется объектом System. Windows.Forms.TreeNode, полученным непосредственно из MarshalByRefObject. Вот некоторые интересные свойства TreeNode.

public class TreeNode: MarshalByRefObject, ICloneable, ISerializable {

 …

 public Color BackColor { get; set; }

 public bool Checked { get; set; }

 public virtual ContextMenu ContextMenu { get; set; }

 public virtual ContextMenuStrip ContextMenuStrip { get; set; }

 public Color ForeColor { get; set; }

 public int ImageIndex { get; set; }

 public bool IsExpanded { get; }

 public bool IsSelected { get; }

 public bool IsVisible { get; }

 public string Name { get; set; }

 public TreeNode NextNode { get; }

 public Font NodeFont { get; set; }

 public TreeNodeCollection Nodes { get; }

 public TreeNode PrevNode { get; }

 public string Text { get; set; }

 public string ToolTipText { get; set; }

…

}

Как видите, каждому узлу TreeView можно назначить изображение, цвет, шрифт, подсказки и контекстное меню. Кроме того, TreeNode предлагает члены, позволяющие перейти к следующему (или предыдущему) TreeNode. С учетом этого рассмотрите начальную реализацию BuildCarTreeView().

private void BuildCarTreeView() {

**// TreeView не отображается, пока не созданы все узлы.**

 treeViewCars.BeginUpdate();

**// Очистка TreeView от текущих узлов.**

 treeViewCars.Nodes.Clear();

**// Добавление TreeNode для каждого объекта Car из List‹›.**

 foreach (Car с in listCars) {

**// Добавление текущего Car в виде узла высшего уровня.**

  treeViewCars.Nodes.Add(new TreeNode(cpetName));

**// Получение только что добавленного Car для построения**

**// двух подчиненных узлов на основе скорости и**

**// внутреннего объекта Radio.**

  treeViewCars.Nodes[listCars.IndexOf(с)].Nodes.Add(new TreeNode(string.Format("Скорость: {0}", с.currSp.ToString())));

  treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(new TreeNode(string.Format("Любимое радио: {0} FM", с.r.favoriteStation)));

 }

**// Отображение TreeView.**

 treeViewCars.EndUpdate();

}

Здесь создание узлов TreeView происходит между вызовами BeginUpdate() и EndUpdate(). Это удобно тогда, когда заполняется "массивный" объект TreeView, содержащий много узлов, поскольку тогда этот элемент управления не отображает свои элементы, пока вы не закончите заполнение коллекции Nodes. В этом случае конечный пользователь не замечает того, что обработка элементов TreeView происходит постепенно.

Узлы высшего уровня добавляются в TreeView с помощью простого просмотра содержимого типа List‹› и вставки нового объекта TreeNode в коллекцию Nodes типа TreeView. После добавления узла высшего уровня этот узел извлекается из коллекции Nodes (с помощью индексатора типа) для добавления подчиненных узлов (которые также представляются объектами TreeNode). Как вы можете догадаться, чтобы добавить подчиненный узел к текущему узлу, нужно с помощью свойства Nodes просто пополнить его внутреннюю коллекцию узлов.

Следующей нашей задачей при работе с этой страницей TabControl будет подсветка (с помощью свойства BackColor) выбранного в настоящий момент узла и отображение информации о выбранном элементе (а также о его родительском и подчиненном узлах) в поле элемента Label. Все этого можно сделать с помощью окна свойств, обработав событие AfterSelect элемента управления TreeView. Это событие генерируете после того, как пользователь выбирает узел с помощью щелчка мыши или клавиш навигации. Вот полная реализация обработчика события AfterSelect.

private void treeViewCars\_AfterSelect(object sender, TreeViewEventArgs e) {

 string nodeInfo = "";

**// Построение строки с информацией о выбранном узле.**

 nodeInfо = string.Format("Вы выбрали: {0}\n", e.Node.Text);

 if (e.Node.Parent != null) nodeInfo += string.Format("Рoдительский узел: {0}\n", e.Node.Parent.Text);

 if (e.Node.NextNode != null) nodeInfo += string.Format("Следующий узел: {0}", e.Node.NextNode.Text);

**// Вывод информации и подсветка узла.**

 lblNodeInfo.Text = nodeInfo;

 e.Node.BackColor = Color.AliceBlue;

}

Поступающий объект TreeViewEventArgs имеет свойство Node, которое возвращает объект TreeNode, представляющий выделенный узел. Вы можете извлечь имя узла (с помощью свойства Text), как и имена его родительского и следующего узлов (с помощью свойств Parent/NextNode). Обратите внимание на то, что здесь объекты TreeNode, возвращающиеся из Parent/NextNode, явно проверяются на равенство значению null, поскольку пользователь может выбрать первый узел высшего уровня или последний подчиненный узел (если такой проверки не будет, может генерироваться NullReferenceException).

**Добавление графических изображений для узлов**

В завершение нашего обзора типа TreeView давайте добавим в ваш пример три изображения \*.bmp, которые будут обозначать каждый из типов узлов. С этой целью добавьте в окно проектирования MainForm новый компонент ImageList (назначив ему имя ListTreeView). Затем добавьте в проект три новых изображения, представляющих (или хотя бы приближенно напоминающих) автомобиль, радио и "скорость", выбрав Project→Add New Item из меню (можете использовать файлы \*.bmp, предлагаемые вместе с загружаемым программным кодом примеров этой книги). Каждый из этих файлов \*.bmp имеет размер 16×16 пикселей (что устанавливается через окно свойств), так что в рамках TreeView они будут выглядеть достаточно хорошо.

После создания файлов изображений выберите ImageList в окне проектирования формы и поместите эти изображения в свойство Images в том порядке, какой показан на рис. 21.23, чтобы гарантировать возможность правильного назначения ImageIndex (0, 1 или 2) каждому узлу.

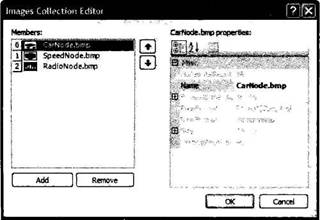


Рис. 21.23. Наполнение ImageList

Вы должны помнить из главы 20, что при добавлении в проект Visual Studio 2006 ресурсов (таких, как точечные рисунки) автоматически обновляется соответствующий файл \*.resx. Таким образом, изображения будут встроены в компоновочный блок без каких бы то ни было дополнительный усилий с вашей стороны. Теперь, используя окно свойств, установите для свойства ImageList элемента управления TreeView значение ImageListTreeView (рис. 21.24).

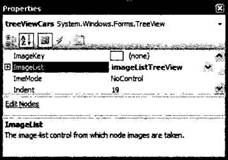


Рис. 21.24. Ассоциация ImageList с TreeView

Наконец, обновите метод BuildCarTreeView(), чтобы при создании каждого TreeNode указывался правильный ImageIndex (с помощью аргументов конструктора).

private void BuildCarTreeView() {

 …

 foreach (Car с in listCars) {

  treeViewCars.Nodes.Add(new TreeNode(c.petName, 0, 0));

  treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(new TreeNode**(string.Format("Скорость: {0}", с.currSp.ToString()), 1, 1)**);

  treeViewCars.Nodes[listCars.IndexOf(с)].Nodes.Add(new TreeNode(**string.Format("Любимое радио: {0} FM", c.r.favoriteStation), 2, 2)**);

 }

 …

}

Обратите внимание на то, что каждый ImageIndex указывается дважды. Причина в том, что TreeNode может иметь два уникальных изображения: одно для отображение тогда, когда узел не выбран, а другое – когда выбран. Чтобы упростить ситуацию, мы указываем одно и то же изображение для обеих возможностей. Так или иначе, обновленный тип TreeView показан на рис. 21.25.

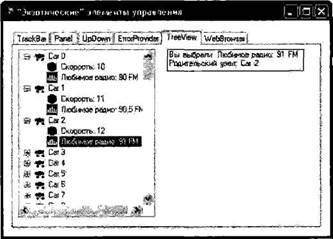


Рис. 21.25. Элемент управления TreeView с рисунками

**Элемент WebBrowser**

На последней странице в этом примере будет использоваться элемент управления System.Windows.Forms.WebBrowser, который появился только в .NET 2.0*.*Этот элемент управления представляет собой окно мини-обозревателя Web, встраиваемого в любой тип Form и обладающего очень широкими возможностями настройки. Как и следует ожидать, этот элемент управления определяет свойство Url, которому может быть присвоено любое действительное значение URI (Uniform Resource Identifier – унифицированный идентификатор ресурса), формально представляемое типом System.Uri. На вкладку WebBrowser добавьте элементы управления WebBrowser (с настройками по вашему выбору), TextBox (для ввода адреса URL) и Button (для выполнения HTTP-запросов). На рис. 21.26 показан вид соответствующего окна в режиме выполнения в момент назначения свойству Url значения http://www.intertechtraining.com (да, это именно то, о чем вы сейчас подумали – беспардонная реклама компании, в которой я работаю).



Рис. 21.26. Элемент WebBrowser, демонстрирующий домашнюю страницу Intertech Training

Чтобы WebBrowser отображал поступающие данные HTTP-запроса, достаточно указать подходящее значение для свойства Url, как это делается в следующем обработчике событий click кнопки Переход.

private void btnGO\_Click(object sender, EventArgs e) {

**// Установка URL в соответствии со значением,**

**// указанным для элемента TextBox страницы.**

 myWebBrowser.Url = new System.Uri(txtUrl.Text);

}

На этом мы завершим наше обсуждение элементов управления из пространства имен System.Windows.Forms. И хотя были рассмотрены не все элементы пользовательского интерфейса, у вас не должно возникнуть проблем при самостоятельном исследовании остальных элементов. Так что теперь давайте перейдем к обсуждению возможности создания *пользовательских*элементов управления Windows Forms.

**Исходный код.**Проект ExoticControls размещен в подкаталоге, соответствующем главе 21.

**Создание пользовательских элементов управления Windows Forms**

Платформа .NET предлагает для разработчиков очень простой способ создания пользовательских элементов интерфейса. В отличие от (теперь уже считающихся устаревшими) элементов управления ActiveX, для элементов управления Windows Forms не требуется громоздкая инфраструктура COM или сложное управление памятью. Вместо этого разработчику нужно просто создать новый класс, получающийся из UserControl, и наполнить этот тип любыми подходящими свойствами, методами и событиями. Для иллюстрации этого процесса мы с помощью Visual Studio 2005 построим пользовательский элемент управления, назвав его CarControl.

**Замечание.**Как и в случае любого .NET-приложения, вы имеете возможность построить любой пользовательский элемент управления Windows Forms "вручную", используя только текстовый редактор и компилятор командной строки. Как вы вскоре убедитесь, элементы управления содержатся в компоновочных блоках \*.dll, поэтому вы должны указать опцию /target:dll компилятора csc.exe.

Начните с запуска Visual Studio 2005 и выберите,для нового проекта рабочее пространство Windows Control Library, указав имя CarControlLibrary (рис 21.27).

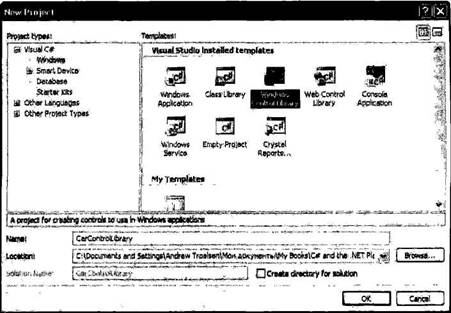


Рис. 21.27. Создание нового рабочего пространства Windows Control Library

После создания проекта переименуйте исходный C#-класс в CarControl. Как и в случае проекта Windows Application, ваш пользовательский элемент управления будет скомпонован из двух классов. Файл \*.Designer.cs содержит программный код, генерируемый инструментами проектирования, а первичный парциальный класс определяет тип, получающийся из System.Windows.Forms.UserControl.

namespace CarControlLibrary {

 public partial class CarControl: **UserControl**{

  public CarControl() {

   InitializeComponent();

  }

 }

}

Перед тем как двигаться дальше, давайте рассмотрим общую картину того, что мы хотим получить. Тип CarControl отвечает за анимацию серии точечных рисунков, которые будут изменяться в зависимости от внутреннего состояния автомобиля. Если текущая скорость автомобиля значительно ниже предельной скорости, для CarControl циклически будут отображаться три точечных рисунка, представляющие безопасное движение автомобиля. Если текущая скорость всего на 10 км/ч ниже максимальной, для CarControl используется цикл из четырех рисунков, где четвертый рисунок изображает автомобиль, медленно распадающийся на части. Наконец, если автомобиль превысит максимальную скорость, циклы CarControl будет состоять из пяти рисунков, где пятый рисунок изображает "обреченный" автомобиль.

**Создание изображений**

В соответствии с представленным выше проектом первым делом нужно создать пять файлов \*.bmp для использования в циклах анимации. Если вы хотите создать свои пользовательские изображения, выберите пункт меню Project→Add New Item и укажите пять новых файлов точечных изображений. Если вы не хотите демонстрировать свои художественные способности, можете использовать изображения, предлагаемые с исходным кодом этого примера (но имейте в виду, что я не считаю себя большим специалистом в области художественной графики!). Первые три изображения (Lemon1.bmp, Lemon2.bmp и Lemon3.bmp) демонстрируют вполне безопасное и аккуратное движение автомобиля по дороге. Другие два изображения (AboutToBlow.bmp и EngineBlown.bmp) представляют автомобиль, приближающийся к максимальному верхнему пределу скорости, и его "безвременную кончину".

**Создание пользовательского интерфейса режима проектирования**

Следующим шагом является использование редактора режима проектирования для типа CarControl. Вы увидите нечто подобное окну проектирования формы, в котором будет изображена клиентская область разрабатываемого вами элемента управления. С помощью окна Toolbox добавьте тип ImageList для хранения точечных рисунков (присвойте этому типу имя carImages), тип Timer (с именем imageTimer) для управления циклом анимации и PictureBox (с именем currentImage) для хранения текущего изображения.

Не беспокойтесь о размерах и размещении типа PictureBox, поскольку характеристики этого элемента в CarControl предполагается задать программно. Однако не забудьте установить в окне свойств значение StretchImage для свойства SizeMode элемента PictureBox. На рис. 21.28 показан желательный результат описанных выше действий.

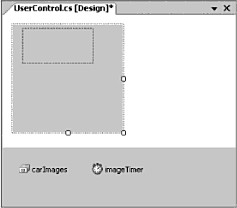


Рис. 21.28. Создание GUI режима проектирования

Затем, используя окно свойств, настройте коллекцию images типа ImageList. добавив рисунки в список. При этом соответствующие элементы нужно добавлять в список последовательно (Lemon1.bmp, Lemon2.bmp, Lemon3.bmp, AboutToBlow.bmp и EngineBlown.bmp), чтобы гарантировать правильную последовательность цикла анимации. Также следует учитывать то, что по умолчанию ширина и высота файлов \*.bmp, вставляемых в Visual Studio 2005, равна 47×47 пикселей. Поэтому ImageSize для ImageList тоже следует установить равным 47×47 (иначе вы получите несколько искаженное изображение). Наконец, настройте свой тип Timer так, чтобы его свойство Interval было равно 200, и он был изначально отключен.

**Реализация CarControl**

После этой подготовительной работы по созданию пользовательского интерфейса вы можете приступить к реализации членов типа. Сначала создайте новый общедоступный перечень AnimFrames, который будет иметь члены, представляющие каждый элемент из ImageList. Этот перечень будет использоваться для определения текущего фрейма, предназначенного для визуализации в PictureBox.

**// Вспомогательный перечень для изображений**.

public enum AnimFrames {

 Lemon1, Lemon2, Lemon3,

 AbоutТоBlow, EngineBlown

}

Тип CarControl поддерживает достаточно большой набор приватных данных, необходимых для представления программной логики анимации. Вот краткое описание каждого из членов.

public partial class CarControl: UserControl {

**// Данные состояния.**

 private AnimFrames currFrame = AnimFrames.Lemon1;

 private AnimFrames currMaxFrame = AnimFrames.Lemon3;

 private bool IsAnim;

 private int currSp = 50;

 private int maxSp = 100;

 private string carPetName= "Lemon";

 private Rectangle bottomRect = new Rectangle();

 public CarControl() {

  InitializeComponent();

 }

}

Как видите, здесь есть данные, представляющие текущую и максимальную скорости, название автомобиля, а также два члена типа AnimFrames. Переменная currFrame используется для указания того, какой из членов ImageList следует отобразить. Переменная currMaxFrame используется для обозначения текущего верхнего предела в ImageList (напомним, что в цикле анимации CarControl используются от трех до пяти изображений, в зависимости от скорости автомобиля). Элемент данных IsAnim используется для определения того, что автомобиль в настоящий момент находится в режиме использования анимации. Наконец, член Rectangle(bottomRect) используется для представления нижней части области CarControl. Позже в этой части элемента управления будет отображаться название автомобиля.

Чтобы разделить CarControl на две прямоугольных области, создайте приватную вспомогательную функцию с именем StretchBox(). Задачей этого члена будет вычисление правильных размеров члена bottomRect и гарантия того, что элемент PictureBox будет растянут на верхние примерно две трети поверхности типа CarControl.

private void StretchBox() {

**// Конфигурация окна изображения.**

 currentImage.Top = 0;

 currentImage.Left = 0;

 currentImage.Height = this.Height – 50;

 currentImage.Width = this.Width;

 currentImage.Image = carImages.Images[(int)AnimFrames.Lemon1];

**// Выяснение размеров нижнего прямоугольника.**

 rect.bottomRect.X = 0;

 bottomRect.Y = this.Height – 50;

 bottomRect.Height = this.Height – currentImage.Height;

 bottomRect.Width = this.Width;

}

После установки размеров каждого прямоугольника в рамках конструктора, заданного по умолчанию, вызывается StretchBox().

public CarControl() {

 InitializeComponent();

 StretchBox();

}

**Определение пользовательских событий**

Тип CarControl обеспечивает поддержку двух событий, отправляемых содержащей тип форме в зависимости от текущей скорости автомобиля. Первое событие, AboutToBlow, генерируется тогда, когда скорость CarControl приближается к верхнему пределу. Событие BlewUp отправляется контейнеру тогда, когда текущая скорость становится больше позволенного максимума. Каждое из этих событий использует пользовательский делегат (CarEventHandler), который может содержать адрес любого метода, возвращающего void и получающего System.String в качестве параметра. Мы обработаем эти события чуть позже, a пока что добавьте к группе открытых элементов CarControl следующие члены.

**// События и пользовательский делегат Car.**

public **delegate** void CarEventHandler(string msg);

public **event** CarEventHandler AboutToBlow;

public **event** CarEventHandler BlewUp;

**Замечание**. Напомним, что "настоящий и полноценный" делегат (см. главу 8) должен указать два аргумента, первым из которых должен быть System.Object (представляющий отправителя), а вторым – тип, производный от System.EventArgs. Однако для нашего примера вполне подойдет и предложенный выше делегат.

**Определение пользовательских свойств**

Как и любой другой тип класса, элемент управления может определять набор свойств, с помощью которых внешние объекты смогут выяснить (или изменить) состояние этого элемента. Нам понадобится определить только три свойства. Сначала рассмотрим свойство Animate. Это свойство включает или отключает тип Timer.

**// Используется для конфигурации внутреннего типа Timer.**

public bool Animate {

 get { return IsAnim; }

 set {

  IsAnim = value;

  imageTimer.Enabled *=*IsAnim;

 }

}

Свойство PetName выполняет то, что и следует ожидать, исходя из его имени, и не требует подробных комментариев. Однако заметьте, что при установке пользователем соответствующего имени выполняется вызов Invalidate(), чтобы это имя CarControl отобразилось в нижней прямоугольной области элемента управления (сам этот шаг будет сделан чуть позже).

**// Выбор имени машины.**

public string PetName {

 get { return carPetName; }

 set {

  CarPetName = value;

  Invalidate();

 }

}

Далее, у нас есть свойство Speed. Вдобавок к простому изменению члена currSp, свойство Speed – это элемент, "стимулирующий" генерирование событий AboutToBlow и BlewUp, в зависимости от текущей скорости CarControl. Вот как выглядит соответствующая программная логика.

**// Проверка currSp и currMaxFrame и генерирование событий.**

public int Speed {

 get { return currSp; }

 set {

**// В пределах безопасной скорости?**

  if (currSp ‹= maxSp) {

   currSp = value;

   currMaxFrame = AnimFrames.Lemon3;

  }

**// Вблизи взрывоопасной ситуации?**

  if ((maxSp – currSp) ‹= 10) {

   if (AboutToBlow != null) {

    AboutToBlow("Чуть помедленнее, парень!");

    currMaxFrame = AnimFrames.AboutToBlow;

   }

  }

**// Превышаем?**

  if (currSp ›= maxSp) {

   currSp = maxSp;

   if (BlewUp != null) {

    BlewUp("М-да… тебе крышка… ");

    currMaxFrame = AnimFrames.EngineBlown;

   }

  }

 }

}

Как видите, если текущая скорость становится лишь на 10 км/ч ниже максимальной допустимой скорости, вы генерируете событие AboutToBlow и сдвигаете верхний предел фреймов анимации к значению AnimFrames.AboutToBlow. Если пользователь превышает возможности вашего автомобиля, вы генерируете событие BlewUp и сдвигаете верхнюю границу фреймов к AnimFrames.EngineBlown. Если скорость ниже максимальной, верхний предел фреймов остается равным AnimFrames.Lemon3.

**Контроль анимации**

Следующей задачей является обеспечение гарантий того, что тип Timer сместит текущий фрейм визуализации в рамках PictureBox. Снова напомним, что число фреймов в цикле анимации зависит от текущей скорости автомобиля. Необходимость изменений изображений в PictureBox возникает только тогда, когда свойство Animate равно true (истина). Начните с обработки события Tick для типа Timer, используя следующий программный код.

private void imageTimer\_Tick(object sender, EventArgs s) {

 if (IsAnim) currentImage.Image = carImages.Images[(int)currFrame];

**// Сдвиг фрейма.**

 int nextFrame = ((int)currFrame) + 1;

 currFrame = (AnimFrames)nextFrame;

 if (currFrame › currMaxFrame) currFrame = AnimFrames.Lemon1;

}

**Отображение названия**

Чтобы завершить создание элемента управления, вы должны отобразить название автомобиля. Для этого обработайте событие Paint для CarControl и в рамках обработчика этого события отобразите PetName вашего CarControl в нижней прямоугольной области клиента.

private void CarControl\_Paint(object sender, PaintEventArgs e) {

**// Отображение названия в нижнем прямоугольнике.**

 Graphics g = e.Graphics;

 g.FillRectangle(Brushes.GreenYellow, bottomRect);

 g.DrawString(PetName, new Font("Times New Roman", 15), Brushes.Black, bottomRect);

}

На этом начальный этап построения CarControl завершается. Теперь выполните компоновку своего проекта.

**Тестирование типа CarControl**

При запуске или отладке проекта Windows Control Library в Visual Studio 2005 иcпользуется UserControl Test Container (испытательный контейнер пользовательских элементов управления). Это управляемый вариант теперь уже устаревшего ActiveX Control Test Container (испытательный контейнер элементов управления ActiveX). Этот инструмент автоматически загружает ваш элемент управления в окружение испытательного стенда режима проектирования. Как показывает рис. 21.29, этот инструмент позволяет установить для проверки значение любого пользовательского свойства (как и любого наследуемого).

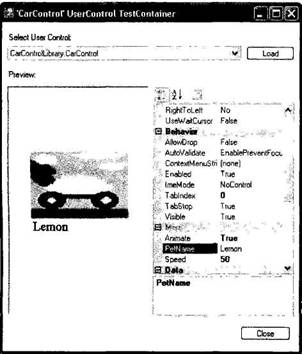


Рис. 21.29. Тестирование CarControl в испытательном контейнере

Установив для свойства Animate значение true (истина), вы увидите цикл анимации CarControl с использованием первых трех файлов \*.bmp. Однако с помощью этой утилиты тестирования вы не сможете обрабатывать события. Чтобы проверить эту возможность вашего элемента интерфейса, нужно построить пользовательскую форму.

**Создание пользовательской формы для CarControl**

Как и в случае любого другого .NET-типа, вы можете использовать свой элемент управления в рамках любого языка, совместимого со средой CLR. Закройте текущее рабочее пространство и создайте новый C#-проект Windows Application с именем CarControlTestForm. Чтобы сослаться на пользовательские элементы управления из Visual Studio 2005, щелкните правой кнопкой мыши в любом месте окна Toolbox и выберите пункт меню Choose Item (Выбрать элемент). Используя кнопку Browse (Просмотр) на вкладке .NET Framework Components (Компоненты .NET), перейдите к своей библиотеке CarControlLibrary.dll. После щелчка на кнопке OK вы обнаружите в панели инструментов новую пиктограмму с названием, конечно же, CarControl.

После этого поместите новый элемент CarControl в окно проектирования формы. Обратите внимание на то, что при этом свойства Animate, PetName и Speed тоже появляются в окне свойств. Точно так же, как и в случае испытательного контейнера, ваш элемент управления "живет своей жизнью" в режиме проектирования. Поэтому, если вы установите для свойства Animate значение true, вы увидите анимацию автомобиля в окне проектирования формы.

Сконфигурировав начальное состояние CarControl, добавьте дополнительные элементы графического интерфейса, которые дозволят увеличивать и уменьшать скорость автомобиля, а также видеть текущую скорость автомобиля и строковые данные, посылаемые генерируемыми событиями (для этого вполне подойдут элементы управления Label). Один из возможных вариантов показан на рис. 21.30.



Рис. 21.30. Графический интерфейс клиента

В предположении о том, что вы создали графический пользовательский интерфейс, аналогичный показанному на рисунке, программный код в рамках вашего типа Form должен быть очень простым (здесь я предполагаю, что вы обработали каждое из событий CarControl с помощью окна свойств).

public partial class MainForm: Form {

 public MainForm() {

  InitializeComponent();

  lblCurrentitSpeed.Text = string.Format("Текущая скорость: {0}", this.myCarControl.Speed.ToString());

  numericUpDownCarSpeed.Value = myCarControl1.Speed;

 }

 private void numericUpDownCarSpeed\_ValueChanged(object sender, EventArgs e) {

**// Предполагается, что минимум NumericUpDown равен 0*,***

**// а максимум - 300.**

  this.myCarControl.Speed = (int)numericUpDownCarSpeed.Value;

  lblCurrentSpeed.Text = string.Format("Текущая скорость: {0}", this.myCarControl.Speed.ToString());

 }

 private void myCarControl\_AboutToBlow(string msg) {

  lblEventData.Text = string.Format("Данные события: {0}", msg);

 }

 private void myCarControl\_BlewUp(string msg) {

  lblEventData.Text = string.Format("Данные событий: {0}", msg);

 }

}

Теперь вы можете запустить ваше приложение клиента на выполнение и проверить его взаимодействие с CarControl. Как видите, задача построения и использования пользовательских элементов управления оказывается достаточно простой, конечно, при условии, что вы уже имеете достаточные знания об ООП, системе типов .NET, GDI+ (т.е. о System.Drawing.dll) и Windows Forms.

Вы сейчас имеете достаточно информации для того, чтобы продолжить исследование процесса разработки элементов управления Windows в .NET, но здесь следует учитывать еще один программный аспект: функциональные возможности режима проектирования. Перед тем как дать точное описание упомянутой проблемы, мы с вами должны обсудить роль пространства имен System.ComponentModel.

**Пространство имен System.ComponentModel**

Пространство имен System.ComponentModel определяет целый ряд атрибутов (и других типов), позволяющих описать то, как должны вести себя ваши элементы управления в режиме проектирования. Например, вы можете указать текстовое описание каждого свойства, определить событие, выбранное по умолчанию, или сгруппировать ряд свойств или событий в пользовательскую категорию, чтобы они отображались вместе в окне свойств Visual Studio 2005. Чтобы выполнить указанные модификации, вы должны использовать атрибуты, показанные в табл. 21.12.

**Таблица 21.12.**Подборка членов System.ComponentModel

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Объекты применения** | **Описание** |
| BrowsableAttribute | Свойства и события | Указывает, должно ли свойство или событие отображаться в окне обозревателя свойств. По умолчанию могут просматриваться все пользовательские свойства и события |
| CategoryAttribute | Свойства и события | Указывает имя категории, к которой относится данное свойство или событие |
| DescriptionAttribute | Свойства и события | Определяет небольшой фрагмент текста, который будет отображаться внизу окна свойств, когда пользователь выбирает данное свойство или событие |
| DefaultPropertyAttribute | Свойства | Указывает свойство, используемое для данного компонента по умолчанию. Это свойство выбирается в окне свойств, когда пользователь выбирает данный элемент управления |
| DefaultValueAttribute | Свойства | Определяет значение по умолчанию для данного свойства, которое будет применено при "переустановке" данного элемента управления в среде разработки |
| DefaultEventAttribute | События | Указывает событие, используемое для данного компонента по умолчанию. Когда программист выполняет двойной щелчок на элементе управления, автоматически генерируется программный код заглушки для события, используемого по умолчанию |

**Совершенствование режима проектирования CarControl**

Чтобы продемонстрировать использование некоторых из этих новых атрибутов, закройте проект CarControlTestForm и снова откройте проект CarControlLibrary. Давайте создадим пользовательскую категорию (назвав ее "Конфигурация машины"), в которой будут отображаться все свойства и события CarControl. Также укажем "дружественное" описание для каждого члена и значение по умолчанию для каждого свойства. Для этого просто обновите каждое из свойств и событий типа CarControl так, чтобы они поддерживали атрибуты [Category], [DefaultValue] и [Description], как показано ниже.

public partial class CarControl: UserControl {

 …

**[Category ("Конфигурация машины"), Description ("Генерируется при приближении к пределу скорости. ")]**

 public event CarEventHandler AboutToBlow;

 ...

**[Category ("Конфигурация машины"), Description("Имя вашей машины"), DefaultValue("Lemon")]**

 public string PetName {…}

 …

}

Теперь позвольте прокомментировать то, что означает присваивание свойству *значения по умолчанию,*поскольку, я уверен, это не то, что вы можете (естественно) предполагать. Упрощенно говоря, атрибут [DefaultValue] не гарантирует, что соответствующее значение элемента данных, представленного данным свойством будет автоматически установлено равным значению по умолчанию. Так, хотя вы и указали значение по умолчанию "Lemon" для свойства PetName, член-переменная carPetName не получит значения "Lemon", пока вы не установите это значение с помощью конструктора типа или синтаксиса инициализации члена (что вы уже на самом деле сделали).

private string carPetName = "Lemon";

Атрибут [DefaultValue] "вступает в игру" тогда, когда программист "переустанавливает" значение данного свойства в окне свойств. Чтобы переустановить свойство в Visual Studio 2005, выберите интересующее вас свойство, щелкните на нем правой кнопкой мыши и в появившемся контекстном меню выберите Reset. Обратите внимание на то, что значение [Description] при этом появляется в нижней панели окна свойств (рис. 21.31).

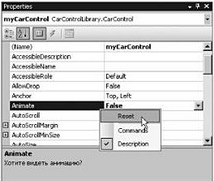


Рис. 21.31. Переустановка свойства

Атрибут [Category] будет проявляться только тогда, когда программист выбирает для просмотра в окне свойств вид, сгруппированный по категориям (в противоположность просмотру по алфавиту, предлагаемому по умолчанию), рис. 21.32.



Рис. 21.32. Пользовательская категория

**Определение выбираемых по умолчанию свойств и событий**

Вдобавок к описаниям членов и группировке членов в категории вы можете настроить свои элементы управления на поддержку поведения, принятого по умолчанию. Так, для элемента управления можно назначить свойство, выбираемое по умолчанию. Для указания такого свойства используется атрибут [DefaultProperty], как показано ниже.

**// Пометка свойства, выбираемого по умолчанию**

**// для данного элемента управления.**

**[DefaultProperty**("Animate")]

public partial class CarControl: UserControl {…}

Тем самым вы гарантируете, что при выборе пользователем этого элемента управления в режиме проектирования в окне свойств автоматически будет выделено свойство Animate. Точно так же для элемента управления указывается выбираемое по умолчанию событие.

**// Пометка события, выбираемого по умолчанию**

**// для данного элемента управления.**

[DefaultEvent("AboutToBlow"), DefaultProperty("Animate")]

public partial class CarControl: UserControl

Тем самым вы гарантируете, что при двойном щелчке пользователя на этом элементе управления в режиме проектирования будет автоматически создан программный код заглушки для выбираемого по умолчанию события (теперь вам должно быть ясно, почему при двойном щелчке на Button автоматически обрабатывается событие Click, при двойном щелчке на Form – событие Load и т.д.).

**Выбор изображений для панели инструментов**

Наконец, непременным атрибутом любого "приличного" пользовательского элемента управления должно быть изображение, представляющее этот элемент управления в окне панели инструментов. В настоящий момент при выборе пользователем CarControl среда разработки покажет этот тип в панели инструментов со стандартной пиктограммой "зубчатки". Чтобы указать пользовательское изображение, первым шагом должно быть добавление в проект нового файла \*.bmp (CarControl.bmp), размеры которого должны быть 16×16 пикселей (устанавливаются с помощью свойств Width и Height). Мы просто используем изображение Car из примера TreeView.

После создания подходящего изображения используйте атрибут [ToolboxBitmap] (который применяется на уровне типа), чтобы назначить это изображение своему элементу управления. Первым аргументом конструктора атрибута должна быть информация типа для элемента управления, а вторым аргументом – имя файла \*.bmp без расширения.

[DefaultEvent("AboutToBlow"), DefaultProperty("Animate"),

**ToolboxBitmap(typeof(CarControl**), "**CarControl**")]

public partial class CarControl: UserControl {...}

Заключительным шагом является выбор значения Embedded Resource для свойства Build Action (с помощью окна свойств), чтобы данные соответствующего изображения были встроены в компоновочный блок (рис. 21.33).

Замечание. Причиной встраивания файла \*.bmp вручную (в отличие от случая использования типа ImageList) является то, что вы не назначаете файл CarControl.bmp элементу пользовательского интерфейса в режиме проектирования, поэтому соответствующий файл \*.resx не получает соответствующих обновленных данных.

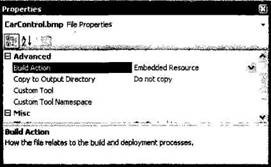


Рис. 21.33. Встраивание ресурсов изображения

После перекомпиляции вашей библиотеки Windows Controls вы можете снова загрузить предыдущий проект CarControlTestForm. Щелкните правой кнопкой на имеющейся пиктограмме CarControl в окне Toolbox и выберите Delete (Удалить).

Затем снова добавьте элемент CarControl в панель инструментов (с помощью щелчка правой кнопкой мыши с последующим выбором Choose Items). На этот раз вы должны увидеть в окне панели инструментов свой пользовательский точечный рисунок (рис. 21.34).



Рис. 21.34. Пользовательская пиктограмма на панели инструментов

На этом наше обсуждение вопросов создания пользовательских элементов управления Windows Forms завершается. Я надеюсь, что этот пример вызвал у вас интерес к разработке пользовательских элементов управления. Здесь была использована стандартная для учебников автомобильная тема, но вы можете представить себе пользовательский элемент управления, отображающий круговую диаграмму, построенную на основе содержимого соответствующей таблицы из базы данных, или элементы управления, расширяющие функциональные возможности стандартных элементов пользовательского интерфейса.

**Замечание.**Чтобы узнать больше о разработке пользовательских элементов управления Windows Forms, обратитесь к книге Matthew MacDonald, *User Interfaces In C#: Windows Forms and Custom Controls*(Apress, 2002).

**Исходный код**. Проект CarControlLibrary размещен в подкаталоге, соответствующем главе 21.

**Создание пользовательских диалоговых окон**

Теперь, когда вы понимаете роль базовых элементов управления Windows Forms и суть процесса построения пользовательских элементов управления, давайте рассмотрим вопрос создания пользовательских диалоговых окон. Здесь хорошей вестью является то, что все уже известное вам о Windows Forms оказывается непосредственно применимым и к случаю программировании диалоговых окон. В общем и целом, создание (и отображение) диалоговых окон ничуть не более сложно, чем добавление в проект новой формы.

В пространстве имен System.Windows.Forms для этого нет специальною базового класса. Диалоговое окно – это просто "специальная" форма. Например, многие диалоговые, окна не позволяют менять свои размеры, поэтому для их свойства FormBorderStyle выбирается значение FormBorderStyle.FixedDialog. Также, в диалоговых окнах свойства MinimizeBox и MaximizeBox обычно равны false (ложь). В этим случае вид диалогового окна вообще является фиксированным, Наконец, если установить значение false для свойства ShowInTaskbar, форме будет запрещено появляться в панели задач Windows XP.

Чтобы продемонстрировать возможности работы с диалоговыми окнами, создайте новое Windows-приложение с именем SimpleModalDialog. Главный тип Form будет поддерживать объект MenuStrip, содержащий пункты меню Файл→Выход и Сервис→Настройка. Постройте этот пользовательский интерфейс и обработайте событие Click для пунктов меню Выход и Настройка. Также определите член-переменную строкового типа (с именем userMessage) в рамках главного типа Form и отобразите соответствующие данные в обработчике события Paint главной формы. Вот как выглядит соответствующий программный код файла MainForm.cs.

public partial class MainWindow: Form {

 private string userMessage = "Сообщение, заданное по умолчанию";

 public MainWindow() {

  InitializeComponent();

 }

 private void exitToolStripMenuItem\_Click(object sender, EventArgs e) {

  Application.Exit();

 }

 private void configureToolStripMenuItem\_Click(obiect sender, EventArgs e) {

**// Этот метод будет реализован чуть позже…**

 }

 private void MainWindow\_Paint(object sender, PaintEventArgs e) {

  Graphics g = e.Graphics;

  g.DrawString(userMessage, new Font("Times New Roman", 24), Brushes.DarkBlue, 50, 50);

 }

}

Теперь добавьте в текущий проект новый объект Form с именем UserMessageDialog.cs с помощью выбора Project→Add Windows Form из меню. Установите для свойств ShowInTaskbar, MinimizeBox и MaximizeBox значения false. Затем постройте пользовательский интерфейс, состоящий из двух типов Button (для кнопок OK и Отмена), одного TextBox (чтобы предоставить пользователю возможность ввести сообщение) и элемента Label с инструкцией. Один возможный вариант показан на рис. 21.35.

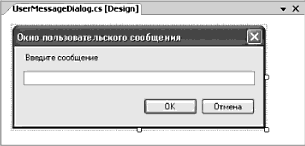


Рис. 21.35. Пользовательское диалоговое окно

Наконец, откройте доступ к значению Text элемента TextBox формы с помощью пользовательского свойства с именем Message.

public partial class UserMessageDialog: Form {

 public UserMessageDialog() {

  InitializeComponent();

 }

 public string Message {

  set { txtUserInput.Text = value; }

  get { return txtUserInput.Text; }

 }

}

**Свойство DialogResult**

В качестве заключительного задания при создании пользовательского интерфейса выберите кнопку OK в окне проектирования формы и найдите свойство DialogResult. Назначьте DialogResult.OK кнопке OK и DialogResult.Cancel – кнопке Отмена. Формально говоря, вы можете назначить свойству DialogResult любое значение из перечня DialogResult.

public enum System.Windows.Forms.DialogResult {

 Abort, Cancel, Ignore, No,

 None, OK, Retry, Yes

}

Но что же означает присваивание значения свойству DialogResult элемента Button? Это свойство может быть назначено для любого типа Button (как и для самой формы), и оно позволяет родительской форме определить, какую из кнопок выбрал конечный пользователь. Для примера измените обработчик меню Сервис→ Настройка в рамках типа MainForm так, как предлагается ниже,

private void configureToolStripMenuIteimClick(object sender, EventArgs e) {

***//*Создание экземпляра UserMessageDialog.**

 UserMessageDialog dlg = new UserMessageDialog();

**// Размещение текущего сообщения в TextBox.**

 dlg.Message = userMessage;

**// Если пользователь щелкнул на кнопке OK, отобразить сообщение.**

 if (DialogResult.OK == dlg.ShowDialog()) {

  userMessage = dlg.Message;

  Invalidate();

 }

**// Лучше, чтобы очистку внутренних элементов выполняло само**

**// диалоговое окно, не дожидаясь сборщика мусора.**

 dlg.Dispose();

}

Здесь UserMessageDialog отображается с помощью вызова ShowDialog(). Этот метод запустит форму в виде модального диалогового окна, а это, как вы знаете, означает, что пользователь не сможет перейти к главной форме, пока диалоговое окно не будет закрыто. После закрытия диалогового окна пользователем (с помощью щелчка на кнопке OK или на кнопке Отмена), форма станет невидимой, но все еще будет оставаться в памяти. Поэтому вы можете запросить у экземпляра UserMessageDialog (с именем dlg) информацию о новом значении Message в том случае, когда пользователь щелкнул на кнопке OK. В этом случае вы отображаете новое сообщение, иначе не делаете ничего.

Замечание. Чтобы отобразить немодальное диалоговое окно (которое позволяет переходить от родительской формы к диалоговой и обратно), следует вызвать Show(), а не ShowDialog().

**Наследование форм**

Одним из наиболее привлекательных аспектов построения диалоговых окон в Windows Forms является *наследование форм.*Вы, несомненно, знаете, что наследование является одним из базовых принципов ООП, который позволяет одному классу расширить функциональность другого. Обычно, когда говорят о наследовании, представляют один тип (например, SportsCar) без графического интерфейса, получающийся из другого типа (например, Car), также не имеющего графического интерфейса. Однако в Windows Forms оказывается вполне возможным получение одной формы из другой, сохранив в процессе наследования элементы базового класса и их реализации.

Наследование на уровне форм является очень мощной технологией программирования, поскольку она позволяет построить базовую форму, обеспечивающую базовые функциональные возможности для целого семейства связанных диалоговых окон. Если связать базовые формы в компоновочном блоке .NET, другие члены вашей команды разработчиков смогут расширять эти типы, используя при этом тот язык .NET, который они предпочитают использовать.

Для примера предположим, что вы хотите создать подкласс класса UserMessageDialog, чтобы в новом диалоговом окне пользователь имел возможность указать, что сообщение должно отображаться курсивом. Для этого выберите Project→Add Windows Form из меню, но на этот раз добавьте новую форму Inherited Form (Наследуемая форма), назначив ей имя ItalicUserMessageDialog.cs (см. рис. 21.36).

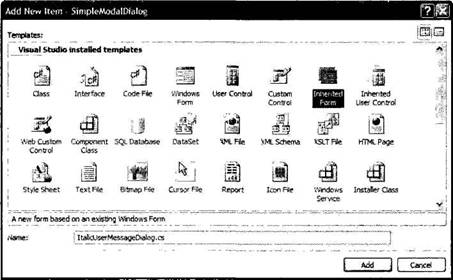


Рис. 21.36. Добавление производной формы

После щелчка на кнопке Add (Добавить) вы увидите окно утилиты Inheritance Picker (Выбор наследования), которая позволяет выбрать форму из вашего текущего проекта или форму из внешнего компоновочного блока (с помощью кнопки Browse). Для данного примера выберите свой уже существующий тип UserMessageDialog. Вы обнаружите, что ваш новый тип Form расширяет ваш тип диалогового окна, а не базовый объект Form непосредственно. Теперь вы можете расширять полученную форму так, как захотите. Для проверки просто добавьте новый элемент управления CheckBox (с именем checkBoxItalic), который будет доступен через свойство, названное Italic.

public partial class ItalicUserMessageDialog: **SimpleModalDialog.UserMessageDialog**{

 public ItalicUserMessageDialog() {

  InitializeComponent();

 }

 public bool Italic {

  set { checkBoxItalic.Checked = value; }

  get { return checkBoxItalic.Checked; }

 }

}

Теперь, имея подкласс базового типа UserMessageDialog, измените MainForm так, чтобы новое свойство Italic можно было использовать. Просто добавьте новый член-переменную типа Boolean для использования при построении объекта Font, представляющего курсивный шрифт, и измените обработчик события Click для меню Сервис→Настройка так, чтобы использовался ItalicUserMessageDialog. Вот как может выглядеть окончательный вариант программного кода.

public partial class MainWindow: Form {

 private string userMessage = "Default Message";

**private bool textIsItalic = false**;

 …

 private void configureToolStripMenuItem\_Click(object sender, EventArgs e) {

**ItalicUserMessageDialog dlg**= **new ItalicUserMessageDialog();**

  dlg.Message = userMessage;

  dlg.Italic = textIsItaliс;

**// Если пользователь щелкнул на OK, отобразить сообщение.**

  if (DialogResult.OK == dlg.ShowDialog()) {

   userMessage = dlg.Message;

   textIsItalic = dlg.Italic;

   Invalidate();

  }

  // Лучше, чтобы очистку внутренних элементов выполняло само

  // диалоговое окно, не дожидаясь сборщика мусора. dlg.Dispose();

 }

 private void MainWindow\_Paint(object sender, PaintEventArgs e) {

  Graphics g *=*e.Graphics;

  Font f = null;

  if (textIsItalic) f = new Font("Times New Roman", 24, FontStyle.Italic);

  else f = new Font("Times New Roman", 24);

  g.DrawString(userMessage, f, Brushes.DarkBlue, 50, 50);

 }

}

**Исходный код.**Проект SimpleModalDialog размещен в подкаталоге, соответствующем главе 21.

**Динамическое позиционирование элементов управления Windows Forms**

Чтобы завершить эту главу, давайте рассмотрим несколько подходов, которые можно использовать для управления размещением элементов управления в форме. Если при создании типа Form вы предполагаете, что элементы управления должны отображаться с использованием *абсолютных позиций,*то это, по сути, означает, что кнопка, размещенная в окне проектирования формы на 10 пикселей ниже и на 10 пикселей правее верхнего левого угла формы, будет там оставаться в течение всей ее "жизни".

При создании формы, содержащей элементы управления пользовательского интерфейса, вы должны решить, должна ли форма позволять изменение размеров ее окна. Обычно главное окно допускает изменение размеров, тогда как диалоговые окна – нет. Напомним, что допустимость изменения размеров формы задается свойством FormBorderStyle. которое может принимать любое из значений перечня FormBorderStyle.

public enum System.Windows.Forms.FormBorderStyle {

 None, FixedSingle, Fixed3D,

 FixedDialog, Sizable,

 FixedToolWindow, SizableToolWindow

}

Предположим, что вы захотели изменить размеры формы. Тогда в связи с содержащимися в форме элементами управления возникают интересные вопросы. Например, если пользователь сделает форму меньше, чем необходимо для отображения всех элементов управления, должны ли эти элементы управления изменять свои размеры (и, возможно, расположение) в соответствии с размерами формы?

**Свойство Anchor**

В Windows Forms свойство Anchor используется для определения относительной фиксированной позиции, в которой всегда должен пребывать данный элемент управления. Каждый производный от Control тип имеет свойство Anchor, которое может принимать любое из значений перечня AnchorStyles, описанных в табл. 21.13.

**Таблица 21.13**. Значения AnchorStyles

|  |  |
| --- | --- |
| **Значение** | **Описание** |
| Bottom | Нижний край элемента управления прикрепляется к нижнему краю контейнера |
| Left | Левый край элемента управления прикрепляется к левому краю контейнера |
| None | Элемент управления не прикрепляется к краям контейнера |
| Right | Правый край элемента управления прикрепляется к правому краю контейнера |
| Top | Верхний край элемента управления прикрепляется к верхнему краю контейнера |

Чтобы закрепить элемент в верхнем левом углу окна, можно связывать соответствующие значения операцией ИЛИ (например, AnchorStyles.Top | AnchorStyles.Left). Целью использования свойства Anchor является указание того, какие расстояния от краев элемента управления до краев контейнера должны быть фиксированы. Например, если задать для кнопки следующее значение Anchor:

**// Закрепление элемента относительно правого края.**

myButton.Anchor = AnchorStyles.**Right;**

то вы гарантируете, что при переопределении размеров формы данная кнопка будет сохранять свое положение относительно правого края формы.

**Свойство Dock**

Другой особенностью программирования Windows Forms является возможность задать *cтыковочное поведение*элементов управления. С помощью свойства Dock элемента управления можно указать, какой стороны (или каких сторон) формы должен касаться данный элемент. Значение, которое вы назначите свойству Dock элемента управления, учитывается вне зависимости от текущих размеров окна формы. Допустимые значения описаны в табл. 21.14.

**Таблица 21.14.**Значения DockStyle

|  |  |
| --- | --- |
| **Значение** | **Описание** |
| Bottom | Нижний край элемента управление стыкуется с нижним краем контейнерного элемента управления |
| Fill | Все края элемента управления стыкуются со всеми краями контейнерного элемента управления, и соответствующим образом изменяется размер |
| Left | Левый край элемента управления стыкуется с левым краем контейнерного элемента управления |
| None | Элемент управления не стыкуется с краем контейнерного элемента управления |
| Right | Правый край элемента управления стыкуется с правым краем контейнерного элемента управления |
| Top | Верхний край элемента управления стыкуется с верхним краем контейнерного элемента управления |

Например, если вы хотите, чтобы данный элемент всегда располагался по левому краю формы, вы должны написать следующее.

**// Этот элемент всегда размещается по левому краю формы,**

**// независимо от текущих размеров формы.**

myButton.**Dock** = DockStyle.**Left**;

Чтобы понять, во что "выливается" установка свойств Anchor и Dock, рассмотрите проект AnchoringControls, который содержится в загружаемом файле примеров для этой книги. После компоновки и запуска этого приложения вы сможете использовать его систему меню для установки различных значений AnchorStyles и DockStyle, чтобы наблюдать изменения, происходящие при этом в поведении типа Button (рис. 21.37).

Не забудьте переопределить размеры формы при изменении свойства Anchor, чтобы выяснить, как на это отвечает Button.

**Исходный код.**Проект AnchoringControls размещен в подкаталоге, соответствующем главе 21,



Рис. 21.37. Приложение AnchoringControls

**Табличное и потоковое размещение элементов**

В .NET 2.0 предлагается еще один способ управления размещением элементов управления в форме – с помощью одного из двух администраторов размещения. Типы TableLayoutPanel и FlowLayoutPanel могут использоваться в области клиента формы с целью управления размещением внутренних элементов управления. Предположим, например, что в окне проектирования формы вы поместили в форму новый элемент FlowLayoutPanel и настроили его на стыковку со всеми краями родительской формы (рис. 21.38).

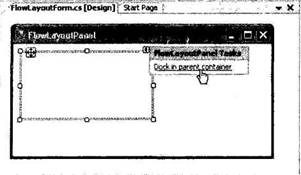


Рис. 21.38. Стыковка FlowLayoutPanel в форме

В режиме проектирования формы добавьте в FlowLayoutPanel десять новых типов Button. Если теперь выполнить приложение, вы заметите, что ваши десять кнопок автоматически распределятся в форме, и это очень напоминает стандартный HTML.

С другой стороны, если вы создаете форму, содержащую TableLayoutPanel, вы можете построить пользовательский интерфейс, который будет поделен на "ячейки" (рис. 21.39).

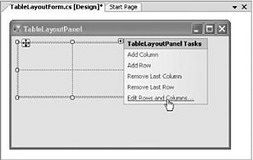


Рис. 21.39. Тип TableLayoutPanel

Если вы выберете пункт Edit Rows and Columns (Редактировать строки и столбцы) из меню встроенного редактора элемента в окне проектирования формы (как показано на рис. 21.39), то вы сможете изменить формат TableLayoutPanel для каждой ячейки (рис. 21.40).

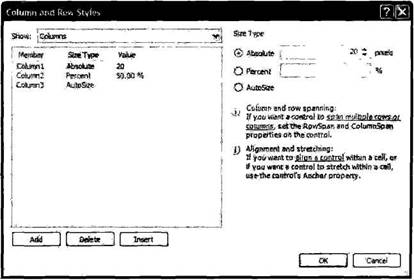


Рис. 21.40. Настройка ячеек типа TableLayoutPanel

Правда, единственной возможностью увидеть результат такой настройки типа TableLayoutPanel является проверка вручную. Предлагаем заинтересованным читателям выполнить это задание самостоятельно.

**Резюме**

Эта глава расширяет ваше понимание пространства имен Windows Forms путем рассмотрения возможностей элементов графического интерфейса пользователя, от самых простых (таких как Label) до "экзотических" (таких как TreeView). После изучения множества типов, соответствующих элементам управления, была рассмотрена задача построения пользовательских элементов управления, включая их интеграцию в среду проектирования.

Также в этой главе вы узнали, как строить пользовательские диалоговые окна и как получить новую форму из уже существующего типа Form, используя возможности наследования форм. Завершилась эта глава кратким обсуждением возможностей закрепления и стыковки элементов управления, а также новых менеджеров размещения .NET 2.0, которые можно использовать для задания нужного расположения элементов графического интерфейса в форме.

**ГЛАВА 22. Доступ к базам данных с помощью ADO.NET**

Если вы не являетесь профессиональным разработчиком видеоигр, вас, наверное, заинтересует тема доступа к базам данных. Как и следует ожидать, платформа .NET определяет целый ряд пространств имен, обеспечивающих взаимодействие с локальными и удаленными хранилищами данных. Эти пространства имен известны под общим названием ADO.NET.

В этой главе, после того как будет "очерчена" роль ADO.NET (в следующем разделе), мы обсудим тему поставщиков данных ADO.NET. Платформа .NET обеспечивает поддержку целого ряда поставщиков данных, каждый из которых оптимизирован для доступа к конкретным системам управления базами данных (Microsoft SQL Server, Oracle, MySQL и т.д.). После того как вы освоите принципы взаимодействия с конкретными поставщиками данных, мы с вами рассмотрим новый шаблон поставщика данных, предлагаемый платформой .NET 2.0. Используя типы из пространства имен System.Data.Common (и файл app.config), можно построить единый программный код, с помощью которого динамически выбирается нужный поставщик данных, без необходимости перекомпиляции и повторной инсталляции приложения.

В оставшейся части главы будет выяснено, как программно взаимодействовать с реляционными базами данных, используя наиболее подходящий для вас поставщик данных. Вы сможете убедиться в том, что ADO.NET обеспечивает два разных уровня взаимодействий с источником данных, часто называемых *связным и несвязным уровнями.*Вы узнаете о роли объектов соединения, объектов команд, объектов чтения данных, адаптеров данных в множества других типов из пространства имен System.Data (В частности, DataSet, DataTable, DataRow, DataColumn, DataView и DataRelation).

**Высокоуровневое определение ADO.NET**

Если вы имеете опыт применения предыдущей модели Microsoft доступа к данным – модели ADO (ActiveX Data Objects – объекты данных ActiveX), основанной на использовании COM, – вы должны понять, что ADO.NET имеет с ADO очень мало общего, кроме букв "A", "D" и "О". Хотя и верно то, что некоторая взаимосвязь между этими двумя системами имеется (например, в каждой из систем используются понятия объекта соединения и объекта команды), некоторые привычные для ADO типы (например, Recordset) в ADO.NET больше не существуют. К тому же, ADO.NET предлагает целый ряд новых типов (например, адаптеры данных), которые не имеют прямых эквивалентов в "классической" модели ADO.

В отличие от классической схемы ADO, которая была разработана, прежде всего для жестко связанных систем клиент/сервер, технология ADO.NET была построена с учетом "разъединенного мира", на базе использования DataSet. Этот тип представляет локальную копию набора связанных таблиц. С помощью DataSet клиентское звено приложения получает возможность читать и обрабатывать его содержимое, будучи отсоединенным от источника данных, чтобы затем с помощью соответствующего адаптера данных направить измененные данные обратно для дальнейшей обработки.

Другим главным отличием ADO.NET от классической схемы ADO является то, что ADO.NET предлагает широкую поддержку XML-представления данных. Сериализация данных, получаемых из хранилища, выполняется (по умолчанию) в формате XML. Поскольку для обмена XML-данными между уровнями взаимодействия часто используется стандартный протокол HTTP, на ADO.NET не влияют ограничения устройств сетевой защиты (брандмауэров).

**Замечание.**В .NET 2.0 сериализация типов DataSet (и DataTable) может выполняться в двоичном формате (с помощью RemotingFormat). Это может оказаться полезным при построении распределенных систем на уровне удаленного взаимодействия .NET (см. главу 18), поскольку двоичные данные оказываются гораздо более компактными, чем данные XML

Но, возможно, самым главным различием между классической схемой ADO и ADO.NET оказывается то, что ADO.NET является управляемой библиотекой программного кода, которая, следовательно, подчиняется правилам, сформулированным для любой управляемой библиотеки. Типы, включенные в ADO.NET, используют тот же протокол управления памятью CLR, ту же систему типов (классы, интерфейсы, перечни, структуры и делегаты) и так же открыты для доступа любому языку .NET.

**Две грани ADO.NET**

Библиотеки ADO.NET могут использоваться в рамках одного из двух концептуально различных способов взаимодействия: на связном или несвязном уровнях. При использовании *связного уровня*ваш программный код непосредственно соединяется с соответствующим хранилищем данных (и отсоединяется от него, когда задачи взаимодействия решены). При использовании ADO.NET в такой форме для взаимодействия с хранилищем данных обычно используются объекты соединения, объекты команд и объекты чтения данных. Позже вы сможете убедиться в том. что объекты чтения данных обеспечивают способ извлечения записей из хранилища данных на основе подхода, допускающего только чтение в режиме однонаправленного доступа.

Несвязный уровень, напротив, позволяет получить набор объектов DataTable (содержащихся в рамках DataSet), функционирующих, как клиентские копии внешних данных. При получении DataSet с помощью соответствующего объекта адаптера данных необходимое вам соединение открывается и закрывается автоматически. Вы можете сами догадаться, что данный подход позволяет быстрее освободить соединение для других вызывавших абонентов. После получения объекта DataSet клиентом этот клиент может просматривать и менять содержимое объекта, не создавая лишней нагрузки на сетевой трафик. Чтобы направить измененные данные обратно в хранилище данных, клиент может снова использовать адаптер данных (в совокупности с множеством подходящих операторов SQL), и после обновления источника данных соединение снова немедленно разрывается.

**Поставщики данных ADO.NET**

ADO.NET не предлагает единого набора типов для связи со всели системами управления базами данных (СУБД). Вместо этого ADO.NET поддерживает множество *поставщиков данных*, каждый из которых оптимизирован для взаимодействия с СУБД конкретного вида. Одним из преимуществ такого подхода является то, что каждый поставщик данных может программироваться с учетом уникальных особенностей соответствующей СУБД. Другим преимуществом является то, что специализированный поставщик данных может соединяться непосредственно с ядром СУБД, без использования промежуточного уровня отображения, размещаемого между связывающимися сторонами.

Упрощенно говоря, *поставщик данных*- это набор типов, определенных в дан-ном пространстве имен и "понимающих", как общаться с конкретным источником данных. Любой используемый нами поставщик данных определяет набор типов, обеспечивающих базовые функциональные возможности. В табл. 22.1 описаны некоторые базовые объекты, их базовые классы (все они определяются в пространстве имен System.Data.Common) и реализованные в них интерфейсы (они определяются в System.Data).

**Таблица 22.1**. Базовые объекты поставщика данных ADO.NET

|  |  |  |  |
| --- | --- | --- | --- |
| **Объект** | **Базовый Класс** | **Реализованные интерфейсы** | **Описание** |
| Connection | DbConnection | IDbConnection | Обеспечивает возможность соединения с хранилищем данных и отключения от него, а также доступ к соответствующему объекту транзакции |
| Command | DbCommand | IDbCommand | Объект команды. Представляет SQL-запрос или имя хранимой процедуры, а также обеспечивает доступ к о6ъекту чтения данных соответствующего поставщика данных |
| DataReader | DbDataReader | IDataReader, IDataRecord | Объект чтения данных. Обеспечивает однонаправленный доступ к данным в режиме "только для чтения" |
| DataAdapter | DbDataAdapter | IDataAdapter, IDbDataAdapter | Объект адаптера данных. Обеспечивает обмен объектами DataSet между вызывающей стороной и местом хранения данных. Содержит набор из четырех внутренних объектов команд, используемых для выборки, вставки, обновления и удаления информации из хранилища данных |
| Parameter | DbParameter | IDataParameter, IDbDataParameter | Объект параметра. Представляет именованный параметр параметризованного запроса |
| Transaction | DbTransaction | IDbTransaction | Объект транзакции. Выполняет транзакцию базы данных |

Хотя имена соответствующих типов для разных поставщиков данных оказываются разными (например, SqlConnection, OracleConnection, OdbcConnection и MySqlConnection), каждый из таких объектов получается из одного и того же базового класса, что предполагает идентичность реализуемых объектами интерфейсов. С учетом этого мы вправе предполагать, что, освоив приемы работы с одним поставщиком данных, освоить остальные поставщики будет совсем просто.

**Замечание.**В соответствии с соглашением о присваивании имен объекты поставщика данных должны иметь префикс, указывающий имя соответствующей СУБД.

На рис. 22.1 показана общая структура поставщика данных ADO.NET. Заметьте, что в представленной диаграмме элемент Компоновочный блок клиента может обозначать практически любое приложение .NET – консольную программу, приложение Windows Forms, Web-страницу ASP.NET, Web-сервис XML, библиотеку программного кода .NET и т.д.

Конечно, в дополнение к объектам, показанным на рис. 22.1, поставщик данных предлагает и другие типы объектов. Однако указанные на рисунке базовые объекты присущи всем поставщикам данных.

**Поставщики данных Microsoft**

В дистрибутив Microsoft .NET 2.0 включен ряд поставщиков данных, в частности для Oracle, SQL Server и ODBC. В табл. 22.2 для поставщиков данных Microsoft ADO.NET указаны пространства имен и содержащие их компоновочные блоки.

**Замечание.**Специального поставщика данных, обращающегося непосредственно к механизму Jet (т.е. к Microsoft Access), нет. Для взаимодействия с файлами данных Access можно использовать поставщик данных OLE DB или ODBC.

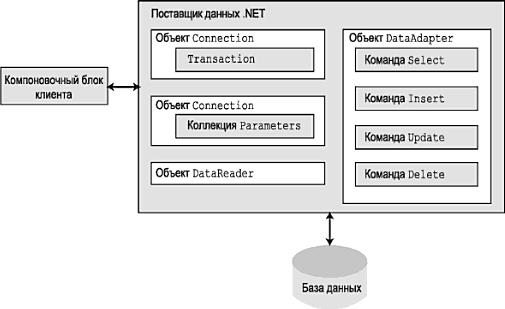


Рис 22.1. Поставщики данных ADO.NET обеспечивают доступ к данным СУБД.

**Таблица 22.2**. Поставщики данных ADO.NET от Microsoft

|  |  |  |
| --- | --- | --- |
| **Поставщик данных** | **Пространство имен** | **Компоновочный блок** |
| OLE DB | System.Data.OleDb | System.Data.dll |
| Microsoft SQL Server | System.Data.SqlClient | System.Data.dll |
| Microsoft SQL Server Mobile | System.Data.SqlServerCe | System.Data.SqlServerCe.dll |
| ODBC | System.Data.Odbc | System.Data.dll |
| Oracle | System.Data.OracleClient | System.Data.OracleClient.dll |

Поставщик данных OLE DB, который скомпонован из типов, определенных в пространстве имен System.Data.OleDb, позволяет получить доступ к данным любого 'хранилища данных, поддерживающего классический протокол OLE DB на основе COM. С помощью этого поставщика данных можно связаться с любой базой данных OLE DB, просто настроив сегмент Provider строки соединения. При этом, однако, следует учитывать то, что поставщик OLE DB в фоновом режиме взаимодействует с различными объектами COM, а это может влиять на производительность приложения. В общем, поставщик данных OLE DB оказывается полезным только в том случае, когда приходится взаимодействовать с СУБД, не определяющей конкретного поставщика данных .NET.

Поставщик данных Microsoft SQL Server предлагает прямой доступ к хранилищам данных Microsoft SQL Server (версий 7.0 и выше) и только к хранилищам данных SQL Server. Пространство имен System.Data.SqlClient содержит типы, используемые поставщиком данных SQL Server и предлагающие, в основном, те же функциональные возможности, что и поставщик данных OLE DB. Ключевым различием является то. что поставщик данных SQL Server действует в обход уровня OLE DB, а это обеспечивает ряд преимуществ с точки зрения производительности системы. Кроме того, поставщик данных Microsoft SQL Server позволяет получить доступ к некоторым уникальным возможностям данной конкретной СУБД.

**Замечание.**Если вас интересуют особенности использования пространств имен System.Data.SqlServerCe, System.Data.Odbc или System.Data.Oracle, за подробностями обратитесь к документации .NET Framework 2.0 SDK.

**Поставщики данных других производителей**

Вдобавок к поставщикам данных, предлагаемым компанией Microsoft, существуют поставщики данных других производителей, предназначенные для самых разных, как свободно доступных, так и коммерческих баз данных. В табл. 22.3 указано, где найти некоторые управляемые поставщики данных, не предлагаемые в комплекте инсталляции Microsoft .NET 2.0 (помните о том, что указанные здесь адреса URL могут измениться).

**Таблица 22.3.**Поставщики данных ADO.NET разных производителей

|  |  |
| --- | --- |
| **Поставщик данных** | **Адрес web-страницы** |
| Firebird Interbase | [http://www.mono-project.com/Firebird\_Interbase](http://www.mono-project.com/firebird_interbase) |
| IBM DB2 | http://www-306.ibm.com/software/data/db2 |
| MySQL | http://dev.mysql.com/downloads/connector/net/1.0.html |
| PostgreSQL | http://www.mono-project.com/PostgreSQL |
| Sybase | http://www.mono-project.com/Sybase |

**Замечание.**Поскольку число поставщиков данных ADO.NET велико, в примерах этой главы будет использоваться поставщик данных Microsoft SQL Server (System.Data.SqlClient). После освоения материала, представленного на страницах этой главы, у вас не должно возникать проблем при использовании ADO.NET для взаимодействия с другими СУБД.

**Дополнительные пространства имен ADO.NET**

В дополнение к пространствам имен .NET, определяющим типы конкретного поставщика данных, библиотеки базовых классов предлагают ряд дополнительных пространств имен, связанных с ADO.NET (табл. 22.4).

Следует понимать, что эта глава не предполагает рассмотрение абсолютно всех типов из каждого пространства имен ADO.NET (для этого потребовалась бы отдельная книга). Но очень важно, чтобы вы поняли суть и возможности типов, предлагаемых в рамках пространства имен System.Data.

Таблица 22.4. Дополнительные пространства имен, имеющие отношение к ADO.NET

|  |  |
| --- | --- |
| **Пространство имен** | **Описание** |
| Misrosoft.SqlServer.Server | Новое пространство имен .NET 2.0; предлагает типы, позволяющие с помощью управляемых языков создавать хранимые процедуры для SQL Server 2005 |
| System.Data | Определяет базовые типы ADO.NET, используемые всеми поставщиками данных |
| System.Data.Common | Содержит типы, совместно используемые поставщиками данных, включая типы соответствующие модели источника поставщика данных .NET 2.0 |
| System.Data.Design | Новое пространство имен .NET 2.0; предлагает различные типы, используемые при настройке пользовательских компонентов данных в режиме проектирования |
| System.Data.Sql | Новое пространство имен .NET 2.0; предлагает типы, позволяющие выявлять экземпляры Microsoft SQL Server, установленные в локальной сети |
| System.Data.SqlTypes | Содержит "собственные" типы данных Microsoft SQL Server. Хотя вы всегда можете использовать соответствующие типы данных CLR, типы SqlTypes оптимизированы специально для работы с SQL Server |

**Типы System.Data**

Пространство имен System.Data является, так сказать, общим знаменателем для всех пространств имен ADO.NET. Вы просто не можете построить приложение ADO.NET, не указав это пространство имен в приложении доступа к данным. Эта пространство имен содержит типы, совместно используемые всеми поставщиками данных ADO.NET, независимо от лежащего в их основе типа хранилища данных. В дополнение к целому ряду исключений (NoNullAllowedException, RowNotInTableException, MissingPrimaryKeyExceeption и т.д.), связанных с доступом к базам данных. System.Data содержит типы, соответствующие как раз-личным примитивам (таблицам, строкам, столбцам, ограничениям и т.д.) базы данных, так и общим интерфейсам, реализуемым объектами поставщика данных, В табл. 22.5 предлагаются описания некоторых базовых типов этого пространства имен, о которых вам следует знать.

Роль пространства имен DataSet, a также DataTable.DataRelation.DataRow и т.д. будет рассмотрена в этой главе позже. Нашей ближайшей задачей будет рассмотрение базовых интерфейсов System.Datа, так сказать, с общей точки зрения. чтобы лучше понять общие функциональные возможности, предлагаемые всеми поставщиками данных. Конкретные детали будут обсуждаться в процессе изложения материала этой главы, а сейчас мы сосредоточимся на общем поведении каждого из имеющихся типов интерфейса.

**Таблица 22.5.**Базовые члены пространства имен System.Data

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| Constraint | Представляет ограничение для данного объекта DataColumn |
| DataColumn | Представляет отдельный столбец в рамках объекта DataTable |
| DataRelation | Представляет отношение "родитель-потомок" между двумя объектами DataTable |
| DataRow | Представляет отдельную строку в рамках объекта DataTable |
| DataSet | Представляет хранимые в памяти данные, скомпонованные на основе любого числа взаимно связанных объектов DataTable |
| DataTable | Представляет табличный блок данных в памяти |
| DataTableReader | Позволяет доступ к DataTable в режиме однонаправленного курсора (только для чтения); этот тип появился в .NET 2.0 |
| DataView | Обеспечивает пользовательское представление для DataTable с использованием сортировки, фильтрации, поиска, редактирования и навигации |
| IDataAdapter | Определяет базовое поведение объекта адаптера данных |
| IDataParameter | Определяет базовое поведение объекта параметра |
| IDataReader | Определяет базовое поведение объекта чтения данных |
| IDbCommand | Определяет базовое поведение объекта команды |
| IDbDataAdapter | Расширяет IDataAdapter с целью получения дополнительных функциональных возможностей объекта адаптера данных |
| IDbTransaction | Определяет базовое поведение объекта транзакции |

**Интерфейс IDbConnection**

Тип IDbConnection реализуется *объектом соединения*поставщика данных, Этот интерфейс определяет множество членов, используемых для настройки соединения с конкретным хранилищем данных, а также позволяет получить объект транзакции поставщика данных. Вот формальное определение IDbConnection.

public interface IDbConnection: IDisposable {

 string ConnectionString { get; set; }

 int ConnectionTimeout { get; }

 string Database { get; }

 ConnectionState State { get; }

 IDbTransaction BeginTransaction();

 IDbTransaction BeginTransaction(IsolationLevel il);

 void ChangeDatabase(string databaseName);

 void Close();

 IDbCommand CreateCommand();

 void Open();

}

**Интерфейс IDbTransaction**

Как видите, перегруженный метод BeginTransaction(), определенный интерфейсом IDbConnection, обеспечивает доступ к *объекту транзакции*поставщика данных. Используя члены, определенные интерфейсам IDbTransaction, вы можете осуществлять программное взаимодействие с сеансом транзакции и соответствующим хранилищем данных.

public Interface IDbTransaction: IDisposable {

 IDbConnection Connection { get; }

 IsolationLevel IsolationLevel { get; }

 void Commit();

 void Rollback();

}

**Интерфейс IDbCommand**

Интерфейс IDbCommand будет реализован *объектом команды* поставщика данных. Как и в других объектных моделях доступа к данным, здесь объекты команд позволяют программно обрабатывать SQL-операторы, хранимые процедуры и параметризованные запросы. Кроме того, с помощью перегруженного метода ExecuteReader() объекты команд обеспечивают доступ к объекту чтения данных поставщика данных,

public Interface IDbCommand: IDisposable {

 string CommandText { get; set; }

 int CommandTimeout { get; set; }

 CommandType CommandType { get; set; }

 IDbConnection Connection { get; set; }

 IDataParameterCollection Parameters { get; }

 IDbTransaction Transaction { get; set; }

 UpdateRowSource UpdateRowSource { get; set; }

 void Cancel();

 IDbDataParameter CreateParameter();

 int ExecuteNonQuery();

 IDataReader ExecuteReader();

 IDataReader ExecuteReader(CommandBehavior behavior);

 object ExecuteScalar();

 void Prepare();

}

**Интерфейсы IDbDataParameter и IDataParameter**

Обратите внимание на то, что свойство Parameters интерфейса IDbCommand возвращает строго типизованную коллекцию, реализующую интерфейс IDataParameterCollection. Этот интерфейс обеспечивает доступ к множеству совместимых с IDbDataParameter типов класса (например, объектов параметров).

public interface IDbDataParameter: IDataParameter {

 byte Precision { get; set; }

 byte Scale { get; set; }

 int Size { get; set; }

}

Интерфейс IDbDataParameter расширяет интерфейс IDataParameter. предлагающий следующие возможности.

public interface IDataParameter **{**

 DbType DbType { get; set; }

 ParameterDirection Direction { get; set; }

 bool IsNullable { get; }

 string ParameterName { get; set; }

 string SourceColumn { get; set; }

 DataRowVersion SourceVersion { get; set; }

 object Value { get; set; }

}

Как видите, интерфейсы IDbDataParameter и IDataParameter позволяют представить параметры SQL-команды (включая хранимые процедуры) в виде специальных объектов параметров ADO.NET, а не в виде сложных строковых литералов.

**Интерфейсы IDbDataAdapter и IDataAdapter**

*Адаптеры данных*используются для извлечения объектов DataSet из хранилища данных и отправки их в хранилище. Интерфейс IDbDataAdapter определяет набор свойств, используемых для поддержки SQL-операторов в операциях выборки, вставки, обновления и удаления данных.

public interface IDbDataAdapter: IDataAdapter {

 IDbCommand DeleteCommand { get; set; }

 IDbCommand InsertCommand { get; set; }

 IDbCommand SelectCommand { get; set; }

 IDbCommand UpdateCommand { get; set; }

}

Кроме этих четырех свойств, адаптер данных ADO.NET наследует поведение, определенное его базовым интерфейсом IDataAdapter. Этот интерфейс определяет ключевую функцию адаптера данных: способность переносить объекты DataSet из приложения вызывающей стороны в хранилище данных и обратно, используя методы Fill() и Update().

Дополнительно интерфейс IDataAdapter позволяет транслировать имена столбцов базы данных в более понятные пользователю дисплейные имена с помощью свойства TableMappings.

public interface IDataAdapter {

 MissingMappingAction MissingMappingAction { get; set; }

 MissingSchemaAction MissingSchemaAction { get; set; }

 ITableMappingCollection TableMappings { get; }

 int Fill(System.Data.DataSet dataSet);

 DataTable[] FillSchema(DataSet dataSet, SchemaType schemaType);

 IDataParameter[] GetFillParameters();

 int Update(DataSet dataSet);

}

**Интерфейсы IDataReader и IDataRecord**

Следующим ключевым интерфейсом является IDataReader, который представляет общие возможности поведения объекта *чтения данных.*Получив IDataReader-тип от поставщика данных ADO.NET, вы можете обратиться к результирующему набору данных в режиме однонаправленного доступа, позволяющего только чтение.

public interface IDataReader: IDisposable, IDataRecord {

 int Depth { get; }

 bool IsClosed { get; }

 int RecordsAffected { get; }

 void Close();

 DataTable GetSchemaTable();

 bool NextResult();

 bool Read();

}

Наконец, вы видите, что IDataReader расширяет интерфейс IDataRecord, определяющий большой набор членов, которые позволяют извлечь из потока строго типизованное значение вместо общего объекта System.Object, предлагаемого перегруженным методом индексатора объекта чтения данных. Вот часть программного кода соответствующих методов GetXXX(), определенных в рамках IDataRecord (весь программный код можно найти в документации .NET Framework 2.0 SDK).

public interface IDataRecord {

 int FieldCount { get; }

 object this[string name] { get; }

 object this[int i] { get; }

 bool GetBoolean(int i);

 byte GetByte(int i);

 char GetChar(int i);

 DateTime GetDateTime(int i);

 Decimal GetDecimal(int i);

 float GetFloat(int i);

 short GetInt16(int i);

 int GetInt32(int i);

 long GetInt64(int i);

 …

 bool IsDBNull(int i);

}

**Замечание.**Перед чтением значения из объекта чтения данных можно использовать метод IDataReader.IsDBNull(), чтобы программно убедиться в том, что соответствующее поле данных не равно null (и не допустить генерирования соответствующего исключения в среде выполнения).

**Интерфейсы и абстрактные поставщики данных**

К этому моменту вы должны иметь более конкретное представление об общих функциональных возможностях, присущих всем поставщикам данных .NET. Напомним, что, хотя имена реализующих типов у разных поставщиков данных оказываются разными, у вас все равно есть возможность программировать эти типы в аналогичной манере – в этом и заключается преимущество полиморфизма, основанного на использовании интерфейсов. Так, если определить метод, имеющий параметр IDbConnection, вы сможете передать ему любой объект соединения ADO.NET.

public static void OpenConnection(IDbConnection cn) {

**// Открытие входного соединения для вызывающей стороны.**

 cn.Open();

}

То же можно сказать и о возвращаемых значениях. Рассмотрите, например, следующую простую программу на C#, которая позволяет вызывающей стороне получить конкретный объект соединения, используя значение пользовательского перечня (здесь предполагается, что вы указали оператор using для System.Data).

namespace ConnectionApp {

 enum DataProvider { SqlServer, OleDb, Odbc, Oracle }

 class Program {

  static void Main(string[] args) {

**// Получение соединения.**

   IDbConnection myCn = GetConnection(DataProvider.SqlServer);

**// Требуется соединение с базой данных SQL Server Pubs.**

   myCn.ConnectionString = "Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs";

**// Открытие соединения с помощью вспомогательной функции.**

   OpenConnection(myCn);

**// Использование соединения и его последующее закрытие.**

   …

   myCn.Close();

  }

  static IDbConnection GetConnection(DataProvider dp) {

   IDbConnection conn = null;

   switch (dp) {

   case DataProvider.SqlServer:

    conn = new SqlConnection();

    break;

   case DataProvider.OleDb:

    conn = new OleDbConnection();

    break;

   case DataProvider.Odbc:

    conn = new OdbcConnection();

    break;

   case DataProvider.Oracle:

    conn = new OracleConnection();

    break;

   }

   return conn;

  }

 }

}

Преимущество использования общих интерфейсов System.Data заключается в том, что в этом случае у вас больше шансов создать более гибкий программный код, который дольше сможет оставаться актуальным. Например, если сегодня вы построите приложение, использующее Microsoft SQL Server, то что вы сможете сделать, если через год-другой руководство вашей компании примет решение перейти на использование Oracle? Если в приложении "жестко" запрограммированы имена типов System.Data.SqlClient, то вам, очевидно, придется сначала их отредактировать, а затем перекомпилировать и повторно инсталлировать компоновочный блок.

**Файлы конфигурации и гибкость приложений**

Для повышения гибкости своих приложений ADO.NET вы можете на стороне клиента использовать файл \*.config, в котором в рамках элемента ‹appSettings› можно указать пользовательские пары ключей и значений. Вспомните из главы 11, что пользовательские даяние можно прочитать программно с помощью типов из пространства имен System.Configuration. Предположим, что в файле конфигурации вы указали строки соединения и поставщика данных так, как показано ниже.

‹configuration›

 ‹appSettings›

  ‹add key=**"provider"**value="SqlServer" /›

  ‹add key**="cnStr"**value="Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/›

 ‹/appSettings›

‹/configuration›

В этом случае вы можете добавить в Main() строки, обеспечивающие программное чтение этих значении. В результате вы, по существу, создадите источник (т.е. генератор) поставщика данных. Вот как может выглядеть соответствующая модификация указанного метода.

static void Main(string[] args) {

**// Чтение значения ключа provider.**

 string dpStr = ConfigurationManager.AppSettings["provider"];

 DataProvider dp = (DataProvider)Enum.Parse(typeof(DataProvider), dpStr);

**// Чтение значения cnStr.**

 string cnStr = ConfigurationManager.AppSettings["cnStr"];

**// Получение соединения.**

 IDbConnection myCn = GetConnection(dp);

 myCn.ConnectionString = cnStr;

}

**Замечание**. Тип ConfigurationManager появился в .NET 2.0. He забудьте также установить ссылку на компоновочный блок System.Configuration.dll и указать using для пространства имен System.Configuration.

Если предыдущий пример преобразовать в библиотеку программного кода .NET (а не в консольное приложение), вы получите возможность создать любое число клиентов, которые могут устанавливать свои соединения, используя различные уровни абстракции. Но чтобы построить действительно полезную библиотеку, реализующую возможности источника поставщика данных, вы должны также использовать объекты команд, объекты чтения данных, адаптеры данных и другие типы, связанные с обработкой данных. Хотя построить такую библиотеку программного кода будет не слишком сложно, для нее потребуется весьма большой по объему программный код. К счастью, что касается .NET 2.0, добрые люди из Редмонда уже встроили все необходимое в библиотеки базовых классов,

**Исходный код.**Проект MyConnectionFactory размещен в подкаталоге, соответствующем главе 22.

**Модель источника поставщика данных .NET 2.0**

В .NET 2,0 предлагается модель источника поставщика данных, с помощью которой, используя обобщенные типы, можно построить единый базовый код для доступа к данным. Более того, используя файлы конфигурации приложения (в частности, их новый раздел ‹connectionStrings›), можно получать строки поставщиков и соединений декларативно, без необходимости перекомпиляции и повторной инсталляции программного обеспечения клиента.

Чтобы разобраться в реализации источника поставщика данных, вспомните из табл. 22.1, что все объекты поставщика данных получаются из одних и тех же базовых классов, определенных в пространстве имен System.Data.Common:

• DbCommand – абстрактный базовый класс для объектов команд;

• DbConnection – абстрактный базовый класс для объектов соединения;

• DbDataAdapter – абстрактный базовый класс для объектов адаптера данных

• DbDataReader – абстрактный базовый класс для объектов чтения данных;

• DbParameter – абстрактный базовый класс для объектов параметров;

• DbTransaction – абстрактный базовый класс для объектов транзакции.

Кроме того, в .NET 2.0 каждый поставщик данных от Microsoft предлагает специальный класс, получающийся из System.Data.Common.DbProviderFactory. Этот базовый класс определяет ряд методов, с помощью которых извлекаются объекты данных, специфичные для данного поставщика. Вот список соответствующих членов DbProviderFactory.

public abstract class DbProviderFactory {

 …

 public virtual DbCommand CreateCommand();

 public virtual DbCommandBuilder CreateCommandBuilder();

 public virtual DbConnection CreateConnection();

 public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();

 public virtual DbDataAdapter CreateDataAdapter();

 public virtual DbDataSourceEnumerator CreateDataSourceEnumeration();

 public virtual DbParameter CreateParameter();

}

Для получения типа, производного от DbProviderFactorу и подходящего для вашего поставщика данных, пространство имен System.Data.Common предлагает тип класса DbProviderFactories. Используя статический метод GetFactory(), можно получить конкретный (и, кстати, уникальный) DbProviderFactory для указанного вами поставщика данных, например:

static void Main(string[] args) {

**// Получение источника поставщика данных SQL.**

 DbProviderFactory sqlFactory = DbProviderFactories.GetFactory("System.Data.SqlClient");

 …

**// Получение источника поставщика данных Oracle.**

 DbProviderFactory oracleFactory = DbProviderFactories.GetFactory("System.Data.OracleClient");

 …

}

Как вы, наверное, и подумали, вместо получения источника с помощью "жестко" закодированной буквальной строки, соответствующую информацию можно прочитать из файла. \*.config клиента (аналогично тому, как это было сделано в предыдущем примере MyConnectionFactory). Чуть позже мы с вами так и сделаем. Но, так или иначе, создав источник своего поставщика данных, вы сможете получить объекты (соединения, команды и т.д.), соответствующие этому поставщику данных.

**Зарегистрированные источники поставщиков данных**

Перед тем как мы с вами рассмотрим вполне законченный пример использования источника поставщика данных ADO.NET, важно обратить внимание на то, что тип DbProviderFactories (в .NET 2.0) позволяет выбрать источники только некоторого подмножества всех возможных поставщиков данных. Список действительных источников поставщиков данных указывается в рамках элемента ‹DbProviderFactories› в файле machine.config вашей инсталляции .NET2.0 (заметим, что значение атрибута invariant идентично значению, передаваемому методу DbProviderFactories.GetFactory()).

‹system.data›

**‹DbProviderFactories›**

  ‹add name="Odbc Data Provider" invariant**="System.Data.Odbc"**description=".Net Framework Data Provider for Odbc" type="System.Data.Odbc.OdbcFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" /›

  ‹add name="OleDb Data Provider" invariant="**System.Data.OleDb"**description=".Net Framework Data Provider for OleDb" type="System.Data.OleDb.OleDbFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" /›

  ‹add name="OracleClient Data Provider" invariant=**"System.Data.OracleClient"**description=".Net Framework Data Provider for Oracle"  type="System.Data.OracleClient.OracleClientFactory, System.Data.OracleClient, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" /›

  ‹add name="SqlClient Data Provider" invariant=**"System.Data.SqlClient"**description=".Net Framework Data Provider for SqlServer"  type="System.Data.SqlClient.SqlClientFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" /›

 ‹/DbProviderFactories›

‹/system.data›

**Замечание.**Если вы хотите использовать модель источника поставщика данных для СУБД, не упомянутой в файле machine.config, то знайте, что подобная модель для множества поставщиков данных, как с открытым программным кодом, так и коммерческих, предлагается дистрибутивом Mono.NET (см. главу 1).

**Рабочий пример источника поставщика данных**

Давайте построим консольное приложение (с именем DataProviderFactory), которое будет печатать имена и фамилии авторов из таблицы Authors базы данных Pubs, создаваемой при установке Microsoft SQL Server (Pubs представляет собой пример базы данных вымышленной издательской компании).

Сначала укажите ссылку на компоновочный блок System.Configuration.dll, добавьте в текущий проект файл арр.config и определите элемент ‹appSettings›. Помните о том, что "официальным" форматом значения поставщика является полное имя пространства имен поставщика данных, а не такое фиктивное имя, как то, которое использовалось выше в примере MyConnectionFactory.

‹configuration›

 ‹appSettings*›*

  ‹!-- Какой поставщик? --›

  ‹add key="provider'' value="System.Data.SqlClient" /›

  ‹!-- Какая строка соединения? *--›*

  ‹add key="cnStr" value="Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/›

 ‹/appSettings›

‹/configuration›

Теперь, когда у вас есть соответствующий файл \*.config, вы можете прочитать из него значения provider и cnStr, используя метод ConfigurationManager.AppSettings(). Значение provider будет передано методу DbProviderFactories.GetFactory(), чтобы получить специфичный для данного поставщика тип источника данных. Значение cnStr будет использоваться для установки свойства ConnectionString типа DbConnection. В предположении о том, что вы: указали using для пространств имен System.Data и System.Data.Common, обновите метод Main() так, как показано ниже.

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Источники поставщиков данных \*\*\*\n");

**// Получение строк соединения и поставщика данных из файла \*.config.**

 string dp*=*ConfigurationManager.AppSettings("provider");

 string cnStr = ConfigurationManager.AppSettings("cnStr");

**// Создание источника поставщика.**

 DbProviderFactory df *=*DbProviderFactories.GetFactory(dp);

**// Создание объекта соединения.**

 DbConnection cn = df.CreateConnection();

 Console.WriteLine("Объект соединения: {0}", cn.GetType().FullName);

 cn.ConnectionString = cnStr;

 cn.Open();

**// Coздание объекта команды.**

 DbCommand cmd = df.CreateCommand();

 Console.WriteLine("Объект команды: {0}", cmd.GetType().FullName);

 cmd.Connection = cn;

 cmd.CommandText = "Select \* From Authors";

**// Вывод данных с помощью объекта чтения данных.**

 DbDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

 Console.WriteLine("Объект чтения данных: {0}", dr.GetType().FullName);

 Console.WriteLine("\n\*\*\*\*\* Авторы в Pubs \*\*\*\*\*");

 while(dr.Read()) Console.WriteLine("-› {0}, {1}", dr["au\_lname"], dr["au\_fname"]);

 dr.Close();

}

Здесь для проверки печатаются полные имена объекта соединения, объекта команды и объекта чтения данных. Запустив это приложение, вы обнаружите, что для чтения данных из таблицы Authors базы данных Pubs был использован поставщик Microsoft SQL Server (рис. 22.2).

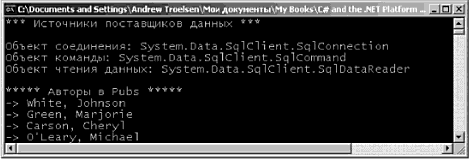


Рис. 22.2. Получение поставщика данных SQL Server с помощью источника поставщика данных .NET 2.0

Далее, если вы измените файл \*.config так, чтобы в нем для поставщика данных было указано System.Data.OleDb (и соответственно обновите строку соединения), как предлагается ниже:

‹configuration›

 ‹appSettings›

  ‹!-- Which provider? --›

  ‹add key="provider" value="System.Data.OleDb"/›

  ‹!-- Which connection string? --›

  ‹add key="cnStr" value="Provider=**SQLOLEDB**;Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs" /*›*

 ‹/appSettings›

‹/configuration›

то вы обнаружите, что теперь в фоновом режиме используются типы System.Data. OleDb (рис. 22.3).



Рис. 22.3. Получение поставщика данных OLE DB с помощью источника поставщика данных .NET 2.0

Конечно, не имея опыта работы с ADO.NET, вы можете иметь слабое представление о том, что именно делают объекты соединения, команды и чтения данных. Но не беспокойтесь до поры до времени о деталях (в конце концов, в этой главе еще немало страниц, а вам только предстоит их прочесть!). На данный момент важно только понять, что в .NET 2.0 вполне возможно построить единый базовый код, который сможет в декларативной форме принимать разных поставщиков данных.

Предложенная модель оказывается очень мощной, но вы должны убедиться в том, что ваш базовый программный код использует только те типы и методы, которые оказываются общими для всех поставщиков. Поэтому при создании общего программного кода ограничьте себя использованием членов, предлагаемых DbConnection.DbCommand и другими типами из пространства имен System.Data. Common. С другой стороны, вполне возможно, что такой "обобщённый" подход не позволит использовать некоторые специфические возможности конкретной СУБД, так что обязательно проверьте работоспособность создаваемого вами программного кода в реальных условиях.

**Элемент ‹connectionStrings›**

В рамках .NET 2.0 файлы конфигурации приложения могут определить новый элемент, названный ‹connectionStrings›. В контексте этого элемента вы можете определять любое число пар имен и значений, которые можно будет прочитать программными средствами, используя индексатор ConfigurationManager.ConnectionStrings. Главным преимуществом этого подхода (в отличие от использования элемента ‹appSettings› и индексатора ConfigurationManager.AppSettings) является то, что в этом случае вы можете определять множество строк соединений для одного приложении в единообразном стиле.

Для примера обновите свой файл арр.config так, как показано ниже (заметьте, что каждая строка соединения здесь задается атрибутами name и connection-String, a не key и value, как в случае ‹appSettings›).

‹configuration›

 ‹appSettings›

  ‹!-- Which provider? --›

  ‹add key= "provider" value="System.Data.SqlClient" /*›*

 ‹/appSettings›

 ‹connectionStrings›

  ‹add name="SqlProviderPubs" connectionString="Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/›

  ‹add name="OleDbProviderPubs" connectionString="Provider=SQLQLEDB.1;Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/›

 ‹/connectionStrings›

‹/configuration›

Теперь обновите метод Main().

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Источники поставщиков данных \*\*\*\n");

 string dp = ConfigurationManager.AppSettings["provider"];

 string cnStr = ConfigurationManager.ConnectionStrings["SqlProviderPubs"].ConnectionString;

 …

}

На этой стадии нашего обсуждения вам уже должно быть ясно, как взаимодействовать с источником поставщика данных .NET 2.0 (и новым элементом ‹connectionStrings›).

**Замечание.**Теперь, когда вы понимаете роль источников поставщиков данных ADO.NET, в остальных примерах этой главы будут использоваться типы из System.Data.SqlClient и "жестко" закодированные строки соединений, чтобы сфокусировать ваше внимание на соответствующих более "узких" задачах обсуждения.

**Исходный код.**Проект DataProviderFactory размещен в подкаталоге, соответствующем главе 22.

**Установка базы данных Cars**

Итак, теперь вам известны основные возможности поставщика данных .NET, и мы можем заняться обсуждением специфики программирования с помощью ADO. NET. Как уже упоминалось, в примерах этой главы будет использоваться Microsoft SQL Server. В русле автомобильной темы, которая используется во всей книге, мы рассмотрим пример базы данных Cars (Автомобили), содержащей три связанные таблицы с именами Inventory (Инвентарь), Orders (Заказы) and Customers (Заказчики).

**Замечание.**Если у вас нет копии Microsoft SOL Server, вы можете загрузить (бесплатную) копию Microsoft SQL Server 2005 Express Edition (<http://lab.msdn.microsoft.com/express>). Хотя этот инструмент и не обладает абсолютно всеми возможностями полной версии Microsoft SQL Server, он позволит вам принять предлагаемую базу данных Cars, При этом учтите то, что примеры данной главы создавались с помощью Microsoft SQL Server, поэтому для выяснения всех проблемных моментов используйте документацию SQL Server 2005 Express Edition.

Чтобы установить базу данных Cars на своей машине, начните с запуска утилиты Query Analyzer (Анализатор запросов), поставляемой в рамках SQL Server. Соединитесь со своей машиной и откройте файл Cars.sql, предлагаемый в папке с исходным кодом примеров для данной главы. Перед тем как выполнить сценарий, убедитесь в том, что путь, указанный в SQL-файле, соответствует вашей инсталляции Microsoft SQL Server. Если необходимо, отредактируйте следующие строки (выделенные полужирным шрифтом).

CREATE DATABASE [Cars] ON (NAME = N'Cars\_Data', FILENAME= **N'С**:**\Program Files\Microsoft SQL Server\MSSQL\Data\Cars\_Data.MDF**', SIZE = 2, FILEGROWTH = 10%)

LOG ON (NAME = N'Cars\_Log', FILENAME = **N'C:\Program Files\Microsoft SQL Server\MSSQL\Data\Cars\_Log.LDF'**, SIZE = 1, FILEGROWTH = 10%)

GO

Теперь выполните сценарий. После этого откройте окно утилиты SQL Server Enterprise Manager. Вы сможете увидеть там три связанные таблицы (с некоторыми уже введенными данными) и одну хранимую процедуру. На рис. 22.4 показаны таблицы, формирующие базу данных Cars.

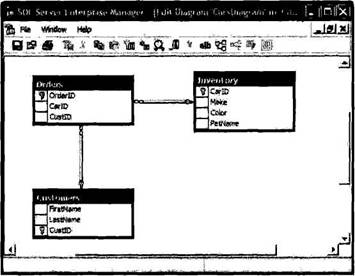


Рис. 22.4. База данных Cars

**Соединение с базой данных в Visual Studio 2005**

Итак, база данных Cars создана, и вы можете установить соединение с этой базой данных из Visual Studio 2005. Это позволит просматривать и редактировать различные объекты базы данных в среде разработки Visual Studio 2005. Используя меню View, откройте окно Server Explorer (Обозреватель серверов). Затем щелкните правой кнопкой мыши на узле Data Connections (Связь с данными) и выберите Add Connection (Добавить соединение) из контекстного меню. В появившемся диалоговом окне выберите в качестве источника данных Microsoft SQL Server. В следующем диалоговом окне выберите имя своей машины из раскрывающегося списка Server Name (Имя сервера) или просто укажите *localhost*а также укажите правильную информацию для входа в систему. Наконец, выберите базу данных Cars из раскрывающегося списка Select or enter a database name (Выбрать или ввести имя базы данных), рис. 22.5.

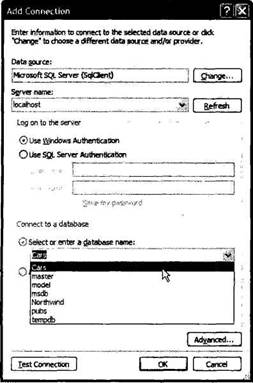


Рис. 22.5. Соединение с базой данных Cars в Visual Studio 2005

После завершения описанной процедуры, в рамках поддерева Data Connections должен появиться узел для Cars. Обратите внимание на то. что здесь же можно увидеть и записи любой таблицы, если щелкнуть на ее имени правой кнопкой мыши и выбрать Show Table Data (Показать данные таблицы) из появившегося контекстного меню (рис. 22.6).

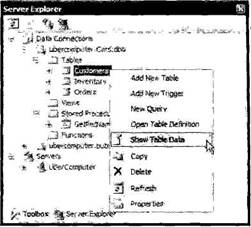


Рис. 22.6. Просмотр данных таблицы

**Связный уровень ADO.NET**

Напомним, что *связный уровень* ADO.NET позволяет взаимодействовать с базой данных, используя объекты соединения, команд и чтения данных вашего поставщика данных. Вы уже использовали эти объекты в предыдущем примере DataProviderFactory, но давайте рассмотрим соответствующий процесс еще раз более подробно. Чтобы соединиться с базой данных и прочитать ее записи с помощью объекта чтения данных, необходимо выполнить следующие шаги

1. Разместить, настроить и открыть объект соединения.

2. Разместить и настроить объект команды, передав ему объект соединения в виде аргумента конструктора или с помощью свойства Connection.

3. Вызвать ExecuteReader() для сконфигурированного объекта команды.

4. Обработать каждую запись, используя метод Read() объекта чтения данных.

Для начала создайте новое консольное приложение с названием CarsDataReader. Нашей целью является открытие соединения (с помощью объекта SqlConnection) а отправка SQL-запроса (с помощью объекта SqlCommand) для получения всех записей из таблицы Inventory базы данных Cars. Затем мы используем SqlDataReader, чтобы напечатать результаты с помощью индексатора типа. Вот соответствующий программный код Main(), за которым следует его анализ.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Забавы с чтением данных \*\*\*\*\*\n");

**// Создание и открытие соединения.**

  SqlConnection cn = new SqlConnection();

  cn.ConnectionString = "uid=sa;pwd=;Initial Catalog=Cars;Data Source=(local)";

  cn.Open();

**// Создание объекта SQL-команды.**

  string strSQL = "Select \* From Inventory";

  SqlCommand myCommand = new SqlCommand(strSQL, cn);

**// Получение объекта чтения данных в стиле ExecuteReader().**

  SqlDataReader myDataReader;

  myDataReader = myCommand.ExecuteReader(CommandBehavior.CloseConnection);

**// Цикл по результатам.**

  while (myDataReader.Read()) {

  Console.WriteLine("-› Марка – {0}, имя – {1}, цвет – {2}.",

   myDataReader["Make"].ToString().Trim(),

   myDataReader["PetName"].ToString().Trim(),

   myDataReader["Color"].ToString().Trim());

  }

**// Поскольку был указан CommandBehavior.CloseConnection,**

**// для соединения нет необходимости явно вызывать Close**().

  myDataReader.Close();

 }

}

**Работа с объектами соединения**

Первым шагом в работе с поставщиком данных является создание соединения с источником данных, для чего мы используем объект соединения (который, как вы помните, получается из Disconnection). Типы соединения .NET получают на вход форматированную *строку соединения,*содержащую набор пар имен и значений, разделенных точками с запятой. Эта информация используется для идентификации машины, с которой требуется установить соединение, параметров безопасности, имени базы данных на удаленной машине и других данных, связанных с конкретным поставщиком данных.

По предыдущему программному коду вы можете заключить, что имя Initial Catalog (исходный каталог) дает ссылку на базу данных, с которой вы пытаетесь соединиться (Pubs, Northwind, Cars и т.д.). Имя Data Source (Источник данных) идентифицирует имя машины, поддерживающей базу данных (для простоты здесь предполагается, что для администраторов локальной системы никакого пароля не требуется).

**Замечание.**Чтобы узнать больше о парах имен и значений для той конкретной СУБД**,**которую используете вы, в документации .NET Framework 2.0 SDK найдите и прочитайте описание свойства ConnectionString объекта соединения для вашего поставщика данных.

После создания строки соединения само соединение с СУБД устанавливается с помощью вызова Open(). В дополнение к ConnectionString, Open() и Close() объект соединения предлагает еще целый ряд членов, которые позволяют настроить дополнительные параметры соединения, например, такие, как время ожидания и свойства транзакций. Описания некоторых членов базового класса DbConnection предлагаются в табл. 22.6.

**Таблица 22.6.**Члены типа DbConnection

|  |  |
| --- | --- |
| **Член** | **Описание** |
| BeginTransaction() | Метод, используемый для начала транзакции |
| ChangeDatabase() | Метод, используемый для смены базы данных при открытом соединении |
| ConnectionTimeout | Доступное только для чтения свойство, возвращающее значение времени  ожидания установки соединения, прежде чем будет сгенерирована ошибка  (значением по умолчанию является 15 секунд). Чтобы изменить значение,  используемое по умолчанию, укажите в строке соединения требуемое  значение Connect Timeout (Например, Сonnect Timeout=30) |
| Database | Свойство, сообщающее имя базы данных, используемой  объектом соединения |
| DataSource | Свойство, сообщающее информации о месте размещения  базы данных, используемой объектом соединения |
| GetSchema() | Метод, возвращающий объект DataSet, который содержит  схему базы данных, полученную от источник данных |
| State | Свойство, устанавливающее текущее состояние соединения в  соответствии со значениями из перечня ConnectionState |

Как видите, свойства типа DbConnection в большинстве своем доступны только для чтения (в силу своей природы) и оказываются полезными только тогда, когда вы хотите получить характеристики соединений в среде выполнения. Чтобы переопределить значение, устанавливаемое по умолчанию, вы должны изменить строку соединения. Например, следующая строка соединения увеличивает время ожидания соединения с 15 до 30 секунд (путем указания соответствующего значения в сегменте Connect Timeout строки соединения).

static void Main(string[] args) {

 SqlConnection cn = new SqlConnection();

 cn.ConnectionString = "uid=sa;pwd=;initial Catalog=Cars;" +

  "Data Source= (local);**Connect Timeout = 30**";

 cn.Open();

**// Новая вспомогательная функция (см. ниже).**

 ShowConnectionStatus(cn);

 …

}

В этом фрагменте программного кода обратите внимание на то, что теперь объект соединения передается в виде параметра новому вспомогательному статическому методу ShowConnectionStatus() класса Program, реализованному так, как показано ниже.

static void ShowConnectionStatus(DbConnection cn) {

**// Отображение информации о текущем объекте соединения.**

 Console.WriteLine("\*\*\*\*\* Информация о соединении \*\*\*\*\*");

 Console.WriteLine("Размещение базы данных: {0}", cn.DataSource);

 Console.WriteLine("Имя базы данных: {0}", cn.Database);

 Console.WriteLine ("Время ожидания: {0}", cn.ConnectionTimeout);

 Console.WriteLine("Состояние соединения: {0}\n", cn.State.ToString());

}

Большинство указанных свойств самоочевидно, Но свойство State все же требует некоторого обсуждения. Этому свойству можно назначить любое значение из перечня ConnectionState

public enum System.Data.**ConnectionState**{

 Broken, Closed,

 Connecting, Executing,

 Fetching, Open

}

но единственными действительными значениями ConnectionState являются ConnectionState.Open и ConnectionState.Closed (остальные члены этого перечня зарезервированы для использования в будущем). Также заметим, что вполне безопасно закрыть соединение, состоянием которого в настоящий момент является ConnectionState.Closed.

**Работа с ConnectionStringBuilder в .NET 2.0**

Работа в программе со строками соединений может оказаться достаточно сложной, например, из-за возможных опечаток, которые не всегда легко обнаружить, В рамках .NET 2.0 все предлагаемые Microsoft поставщики данных ADO.NET поддерживают *объекты построителя строк соединений,*позволяющие создавать пары имен и значений с помощью строго типизованных свойств. Рассмотрим следующую модификацию метода Main().

static void Main(string[] args) {

**// Создание строки соединения с помощью объекта построителя.**

 SqlConnectionStringBuilder cnStrBuilder = new SqlConnectionStringBuilder();

 cnStrBuilder.UserID = "sa";

 cnStrBuilder.Password = "";

 cnStrBuilder.InitialCatalog = "Cars";

 cnStrBuilder.DataSource = "(local)";

 cnStrBuilder.ConnectTimeout =30;

 SqlConnection cn = new SqlConnection();

 cn.ConnectionString = cnStrBuilder.ConnectionString;

 cn.Open();

 ShowConnectionStatus(cn);

 …

}

В этом варианте программного кода создается экземпляр SqlConnectionStringBuilder, устанавливаются соответствующие свойства и с помощью свойства ConnectionString получается внутренняя строка. Заметьте, что здесь используется конструктор типа, заданный по умолчанию. Можно также создать экземпляр объекта построителя строк соединения для поставщика данных, передав уже существующую строку соединения в качестве исходной (это может оказаться полезно тогда, когда соответствующие значения считываются динамически ив файла app.config). Получив такой объект с начальными строковыми данными, вы можете изменить пары имен и значений с помощью соответствующих свойств, например:

static void Main(string[] args) {

 Console.WriteLine("\*\*\* Забавы с чтением данных \*\*\*\n");

**// Предположим, что строка cnStr получeна из файла \*.config.**

 string cnStr = "uid=sa;pwd=;Initial Catalog=Cars;" +

  "Data Source=(local);Connect Timeout=30";

 SqlConnectionStringBuilder cnStrBuilder = new SqlConnectionStringBuilder(cnStr);

 cnStrBuilder.UserID = "sa";

 cnStrBuilder.Password = "";

 caStrBuilder.InitialCatalog = "Cars";

 cnStrBuilder.DataSource = "(local)";

**// Изменение значения времени ожидания**.

 cnStrBuilder.ConnectTimeout = 5;

 …

}

**Работа с объектами команд**

Теперь, когда вы понимаете роль объекта соединения, мы выясним, как предъявить SQL-запрос базе данных. Тип SqlCommand (который получается из DbCommand) является объектом представлением SQL-запроса, имени таблицы или хранимой процедуры. Вид соответствующей команды указывается c помощью свойства CommandTyре, которое может принимать любое значение из перечня CommandType.

public enum System.Data.CommandType {

 StoredProcedure,

 TableDirect,

 Text // **Значение, используемое по умолчанию.**

}

При создании объекта команды вы можете указать SQL-запрос или в качестве параметра конструктора, или напрямую через свойство CommandText. Также при создании объекта команд вы должны указать соединение, которое будет при этом использоваться. Это можно сделать либо через параметр конструктора, либо с помощью свойства Connection.

static void Main(string[] args) {

 SqlConnection cn = new SqlConnection();

 …

**// Создание объекта команды с помощью аргументов конструктора.**

 string strSQL = "Select \* From Inventory";

 SqlCommand myCommand = new SqlCommand(strSQL, cn);

**// Создание другого объекта команды с помощью свойств.**

 SqlCommand testCommand = new SqlCommand();

 testCommand.Connection = cn;

 testCommand.CommandText = strSQL;

 …

}

Следует понимать, что в этот момент вы еще не предъявляете SQL-запрос базе данных Cars непосредственно, а только подготавливаете объект команды для использования в будущем. В табл. 22.7 приводятся описания еще нескольких членов типа DbCommand.

**Таблица 22.7.**Члены типа DbCommand

|  |  |
| --- | --- |
| **Член** | **Описание** |
| CommandTimeout | Читает или устанавливает значение времени ожидания выполнения команды, прежде чем будет сгенерировано сообщение об ошибке. Значением по умолчанию является 30 секунд |
| Connection | Читает или устанавливает значение DbConnection, которое используется данным экземпляром DbCommand |
| Parameters | Получает коллекцию типов DbParameter, используемых для параметризованного запроса |
| Cancel() | Отменяет выполнение команды |
| ExecuteReader() | Возвращает объект DbDataReader поставщика данных для доступа к соответствующим данным режиме однонаправленного чтения |
| ExecuteNonQuery() | Направляет текст команды в хранилище данных |
| ExecuteScalar() | "Облегченная" версия метода ExecuteNonQuery(), предназначенная специально для запросов, возвращающих одиночные данные (например, как при запросе числа записей) |
| ExecuteXmlReader() | В Microsoft SQL Server (2000 и более поздних версий) допускается возможность возвращения набора результатов в формате XML. Данный метод возвращает System.Xml.XmlReader, который позволяет обработать поступающий XML-поток |
| Prepare() | Создает подготовленную (или скомпилированную) версию команды для источника данных. Вы, возможно, знаете, что *готовый к использованию*запрос выполняется быстрее, и это оказывается важно тогда, когда один и тот же запрос требуется выполнять многократно |

**Замечание.**Позже в этой главе будет показано, что в .NET 2.0 у объекта SqlCommand появилось несколько дополнительных членов, упрощающих задачу асинхронного взаимодействия с базами данных.

**Работа с объектами чтения данных**

После создания активного соединения и SQL-команды следующим шагом является предъявление запроса источнику данных. Как вы, наверное, догадываетесь, это можно сделать несколькими способами. Тип DbDataReader (реализующий IDataReader) обеспечивает самый простой и самый быстрый способ получения информации из хранилища данных. Напомним, что объекты чтения данных создают однонаправленный и доступный только для чтения поток данных, возвращающей по одной записи за один раз. Поэтому должно быть вполне очевидно, что объекты чтения данных используются для отправки хранилищу данных только SQL-операторов выборки данных.

Объекты чтения данных оказываются полезными тогда, когда требуется очень быстро просмотреть большой объем данных и при этом нет необходимости заботиться об их представлении в памяти. Например, если вы запрашиваете 20000 записей из таблицы для того, чтобы сохранить их в текстовом файле, то при хранении этой информации в DataSet возникает достаточно большая нагрузка на память. Более выгодным решением оказывается создание объекта чтения данных, который будет обрабатывать каждую запись настолько быстро, насколько это возможно. Но при этом следует учитывать то, что объекты чтения данных (в отличие от объектов адаптера данных, которые мы рассмотрим позже) поддерживают открытое соединение с источником данных, пока вы явно не закроете сеанс связи.

Объекты чтения данных получаются из объекта команды с помощью вызова ExecuteReader(). При вызове этого метода объекту чтения данных можно дополнительно дать инструкцию автоматически закрыть соответствующий объект соединения, указав CommandBehavior.CloseConnection.

В следующем примере использования объекта чтения данных применяется метод Read() для определения момента достижения конца записей (тогда возвращаемое значение оказывается равным false). Для каждой поступающей записи индексатору типа дается указание напечатать информацию о марке автомобиля, его названии и цвете. Также обратите внимание на то, что сразу же после завершении обработки записей вызывается метод Close(), чтобы освободить объект соединения.

static void Main(string[] args) {

 …

**// Получение объекта чтения данных в стиле ExecuteReader().**

 SqlDataReader myDataReader;

 myDataReader = myCommand.ExecuteReader(CommandBehavior.CloseConnection);

**// Цикл по результирующему набору.**

 while(myDataReader.Read()) {

  Console.WriteLine("-› Марка – {0}, название – {1}*,*цвет – {2}",

   myDataReader["Make"].ToString().Trim(),

   myDataReader["PetName"].ToString().Trim(),

   myDataReader["Color"]. ToString().Trim());

 }

 myDataReader.Close();

 ShowConnectionStatus(on);

}

**Замечание.**Обрезка строковых данных здесь используется только для того, чтобы удалить пробелы в конце полей базы данных, и это никак не связано с ADO.NET.

Индексатор объекта чтения данных перегружен, чтобы ему можно было передать либо строку (представляющую имя столбца), либо целое число (соответствующее порядковому номеру столбца). Поэтому вы можете немного "подчистить" соответствующий фрагмент программы (избавившись от необходимости печатания имен) и использовать вариант, показанный ниже (обратите внимание на использование свойства FieldCount).

while (myDataReader.Read()) {

 Console.WriteLine("\*\*\*\*\* Запись \*\*\*\*\*");

 for (int i = 0; i ‹ myDataReader.**FieldCount;**i++) {

  Console.WriteLine("{0} = {1} ", myDataReader.GetName(i), myDataReader.GetValue(i).ToString().Trim());

 }

 Console.WriteLine()*;*

}

После компиляции и запуска этого проекта вы должны увидеть список всех автомобилей из таблицы Inventory базы данных Cars (рис. 22.7).



Рис. 22.7. Объекты чтения данных

**Получение множества наборов результатов с помощью объектов чтения данных**

Объекты чтения данных могут получать от одного объекта команды множество наборов результатов. Например, чтобы получить все строки таблицы Inventory и все строки таблицы Customers, можно указать оба соответствующих SQL-оператора, используя в качестве разделителя точку с запятой.

string theSQL = "Select \* From Inventory:Select \* "from Customers";

Получив объект чтения данных, можно переходить от одного результирующего набора к другому с помощью метода NextResult(). При этом возвращение к первому набору происходит автоматически. Поэтому, чтобы прочитать строки каждой таблицы, можно построить следующую итерационную конструкцию.

do {

 while (myDataReader**.Read()**) {

**// Чтение информации текущего набора результатов.**

 }

} while (myDataReader**.NextResult()**);

Теперь вы должны знать больше о возможностях объектов чтения данных. Эта объекты предлагают и другие возможности, о которых здесь не упоминалось (например, выполнение скалярных и однострочных запросов). Соответствующие подробности можно найти в документации .NET Framework 2.0 SDK.

**Исходный код**. Проект CarsDataReader размещен в подкаталоге, соответствующем главе 22.

**Изменение содержимого таблиц с помощью объектов команд**

Вы только что убедились, что метод ExecuteReader() извлекает объект чтения данных, позволяющий проверить результаты выполнения SQL-оператора Select в однонаправленном и доступном только для чтения потоке. Но если вы хотите применить SQL-команду, в результате которой должна произойти модификация таблицы, вы должны вызвать метод ExecuteNonQuery() соответствующего объекта команды. Этот метод выполняет вставки, обновления и. удаления в соответствии с форматом соответствующей команды.

Чтобы проиллюстрировать возможность модификации существующей базы данных с помощью вызова ExecuteNonQuery(), мы с вами построим новое консольное приложение (CarsInventoryUpdater), предоставляющее пользователю возможность изменения данных таблицы Inventory базы данных Cars. Как и в других примерах, метод Main() здесь отвечает за получение от пользователя инструкций по поводу выполнения конкретных действий, что программно реализуется с помощью оператора switch. Программа разрешает пользователю ввести следующие команды:

• I - вставить новую запись в таблицу Inventory;

• U - обновить существующую запись в таблице Inventory;

• D – удалить существующую запись из таблицы Inventory;

• L – вывести информацию об имеющемся наборе автомобилей, используя объект чтения данных;

• S – показать эти варианты выбора пользователю;

• Q - выйти из программы.

Каждый возможный вариант обрабатывается своим уникальным статическим методом в рамках класса Program. Для полной ясности вот реализация Main(), которая, как кажется, не требует никаких дополнительных комментариев.

static void Main(string[] args) {

 Console.WriteLine ("\*\*\*\*\* Модификатор Inventory для Car *\*\*\*\*\**");

 bool userDone = false;

 string userCommand = "";

 SqlConnection cn = new SqlConnection();

 cn.ConnectionString = "uid=sa;pwd=;Initial Catalog=Cars;" +

  "Data Source=(local);Connect Timeout=30";

 cn.Open();

 ShowInstructions();

 do {

  Console.Write("Введите команду: ");

  userCommand = Console.ReadLine();

  Console.WriteLine();

  switch (userCommand.ToUpper()) {

  case "I":

   InsertNewCar(cn);

   break;

  case "U":

   UpdateCarPetName(cn);

   break;

  case "D":

   DeleteCar(cn);

   break;

  case "L":

   ListInventory(cn);

   break;

  case "S":

   ShowInstructions();

   break;

  case "Q":

   userDone = true;

   break;

  default:

   Console.WriteLine("Некорректные данные! Введите другие");

   break;

  }

 } while (!userDone);

 cn.Close();

}

Метод ShowInstructions() делает то, что и следует ожидать.

private static void ShowInstructions() {

 Console.WriteLine();

 Console.WriteLine("I: добавление новой машины.");

 Console.WriteLine("U: модификация имеющейся машины.");

 Console.WriteLine("D: удаление имеющейся машины.");

 Console.WriteLine("L: список наличных машин.");

 Console.WriteLine("S: вывод инструкций*.*");

 Console.WriteLine(''Q: выход из программы.");

}

Как уже упоминалось, метод ListInventorу() печатает строки таблицы Inventory с помощью объекта чтения данных (соответствующий программный код аналогичен программному коду предыдущего примера CarsDataReader).

private static void ListInventory(SqlConnection cn) {

 string strSQL = "Select \* From Inventory";

 SqlCommand myCommand = new SqlCommand(strSQL, cn);

 SqlDataReader myDataReader;

 myDataReader = myCommand.ExecuteReader();

 while (myDataReader.Read()) {

  for (int i = 0; i ‹ myDataReader.FieldCount; i++) {

   Console.Write("{0} = {1}"; myDataReader.GetNаmе(i), myDataReader.GetValue(i).ToString().Trim());

  }

  Console.WriteLine();

 }

 myDataReader.Close();

}

Итак, текстовый интерфейс пользователя готов, и мы можем теперь перейти к более интересным вещам.

**Вставка новых записей**

Для вставки новой записи в таблицу Inventory нужно (на основе пользовательского ввода) создать SQL-оператор вставки и вызвать ExecuteNonQuery(). Чтобы не загромождать программный код, здесь из него удалена необходимая программная логика try/catch, которая присутствует в загружаемом варианте программного кода примеров для этой книги.

private static void InsertNewCar(SqlConnection cn) {

**// Сбор информации о новой машине.**

 Console.Write("Введите номер машины: ");

 int newCarID = int.Parse(Console.ReadLine());

 Console.Write("Введите марку: ");

 string newCarMake = Console.ReadLine();

 Console.Write("Введите цвет: ");

 string newCarColor = Console.ReadLine();

 Console.Write("Введите название: ");

 string newCarPetName = Console.ReadLine();

**// Создание и выполнение оператора SQL.**

 string sql = string.Format("Insert Into Inventory" +

  "(CarID, Make, Color, PetName) Values" +

  "({0}', '{1}', '{2}', '{3}')",

  newCarID, newCarMake, newCarColor, newCarPetName);

 SqlCommand cmd = new SqlCommand(sql, cn);

 cmd.ExecuteNonQuery();

}

**Замечание.**Вы, возможно, знаете, что построение SQL-операторов с помощью конкатенации строк достаточно рискованно с точки зрения безопасности (вспомните о возможных атаках SQL-инъекции). Здесь этот подход используется только для простоты, поскольку, конечно же, предпочтительнее строить текст команд с помощью параметризованных запросов, обсуждение которых предполагается немного позже.

**Удаление записей**

Удалить существующую запись так же просто, как и вставить новую. Но, в отличие от программного кода для InsertNewCar(), ниже демонстрируется важная возможность применения try/catch для обработки попытки удаления автомобиля, используемого в настоящий момент в процессе оформления заказа для покупателя из таблицы Customers (сама эта таблица будет рассмотрена в этой главе позже).

private static void DeleteCar(SqlConnection cn) {

**// Получение номера машины для удаления и само удаление.**

 Console.Write("Введите номер машины для удаления: ");

 int carToDelete = int.Parse(Console.ReadLine());

 string sql = string.Format("Delete from Inventory where CarID = '{0}'", carToDelete);

 SqlCommand cmd = new SqlCommand(sql, cn);

 try { cmd.ExecuteNonQuery(); } catch {

  Console.WriteLine("Извините, на эту машину оформляется заказ!");

 }

}

**Обновление записей**

Если вы разобрались с программным кодом для DeleteCar() и InsertNewCar(), то и программный код для UpdateCarPetName() не будет для вас сложным (здесь для простоты логика try/catch тоже исключена).

private static void UpdateCarPetName(SqlConnection cn) {

**// Получение номера машины для модификации и ввод нового названия.**

 Console.Write("Введите номер машины для модификации: ");

 string newPetName = "";

 int carToUpdate = carToUpdate = int.Parse(Console.ReadLine());

 Console.Write("Введите новое название: ");

 newPetName = Console.ReadLine();

**// Обновление записи.**

 string sql = string.Format("Update Inventory Set PetName='{0}' Where CarID='{1}'", newPetName, carToUpdate);

 SqlCommand cmd = new SqlCommand(sql, cn);

 cmd.ExecuteNonQuery();

}

На этом создание приложения завершается. На рис. 22.8 показан результат тестового запуска этого приложения.

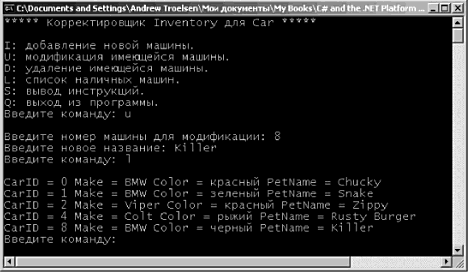


Рис. 22.8. Вставка, обновление и удаление записей c помощью объектов команд

**Работа с объектами параметризованных команд**

Показанная выше программная логика вставки, обновления и удаления работает так, как и ожидается, однако обратите внимание на то, что каждый из SQL-запросов здесь представлен "жестко" закодированными строковыми литералами. Вы, возможно, знаете, что с SQL-параметрами можно обращаться, как с объектами, а не с простыми строками текста, если использовать *параметризованные запросы*Обычно параметризованные запросы выполняются намного быстрее буквальных SQL-строк, поскольку они анализируются только один раз (а не каждый раз, когда SQL-строка присваивается свойству CommandText). Параметризованные запросы также обеспечивают защиту от атак SQL-инъекции (это известная проблема безопасности доступа к данным).

Объекты команд ADO.NET поддерживают коллекцию дискретных типов параметра. По умолчанию эта коллекция пуста, но вы можете добавить в нее любое число объектов параметра, которые должны будут отображаться в "заместитель" параметра в SQL-запросе. Чтобы ассоциировать параметр в SQL-запросе с членом коллекции параметров данного объекта команды, добавьте к текстовому SQL-параметру префикс @ (это работает, как минимум, при использовании Microsoft SQL Server, но такое обозначение поддерживают не все СУБД).

**Указание параметров с помощью типа DbParameter**

Перед тем как приступить к построению параметризованных запросов, мы должны рассмотреть тип DbParameter (который является базовым классом объектов параметров, специфичных для конкретного поставщика данных). Этот класс поддерживает ряд свойств, позволяющих указать имя, размер и тип данных параметра, а также другие его особенности, например направление параметра. В табл. 22.8 описаны некоторые свойства типа DbParameter.

**Таблица 22.8.**Ключевые члены типа DbParameter

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| DbType | Читает или записывает информацию о "родном" типе данных для источника данных, представленную в виде соответствующего типа данных CLR |
| Direction | Читает или записывает значение, указывающее направление потока для данного параметра (только ввод, только вывод, двунаправленное движение, предусмотренное возвращение значения) |
| IsNullable | Читает или записывает значение, являющееся индикатором того, что параметр допускает значения null |
| ParameterName | Читает или устанавливает имя DbParameter |
| Size | Читает или устанавливает максимальный размер данных параметра |
| Value | Читает или устанавливает значение параметра |

Для иллюстрации мы модифицируем предыдущий метод InsertNewCar(), чтобы в нем использовались объекты параметра. Вот как может выглядеть соответствующий программный код.

private static void InsertNewCar(SqlConnection cn) {

 …

**// Обратите внимание на 'заполнители' в SQL-запросе.**

 string sql = string.Format("Insert Into Inventory" +

  "(CarID, Make, Color, PetName) Values" +

  "(@CarID, @Make, @Color, @PetName)");

**// Наполнение коллекции параметров**.

 SqlCommand cmd = new SqlCommand(sql, cn);

 SqlParameter param = new SqlParameter();

 param.ParameterName = "@CarID";

 param.Value = newCarID;

 param.SqlDbType = SqlDbType.Int;

 cmd.Parameters.Add(param);

 param = new SqlParameter();

 param.ParameterName = "@Make";

 param.Value = newCarMake;

 param.SqlDbType = SqlDbType.Char;

 param.Size = 20;

 cmd.Parameters.Add(param);

 param = new SqlParameter();

 param.ParameterName = "@Color";

 param.Value = newCarColor;

 param.SqlDbType = SqlDbType.Char;

 param.Size = 20;

 cmd.Parameters.Add(param);

 param = new SqlParameter();

 param.ParameterName = "@PetName";

 param.Value = newCarPetName;

 param.SqlDbType = SqlDbType.Char;

 param.Size = 20;

 cmd.Parameters.Add(param);

 cmd.ExecuteNonQuery();

}

Хотя при создании параметризованного запроса требуется вводить большой объем программного кода, конечный результат оказывается более выгодным с точки зрения программного использования SQL-операторов. Улучшается также и общая производительность. Вы, конечно, можете использовать предлагаемый подход для всех SQL-запросов, но наиболее полезными параметризованные запросы оказываются при запуске хранимых процедур.

**Замечание**. Здесь для создания объектов параметров использовались различные свойства. Но следует знать и о том, что объекты параметров поддерживают целый ряд перегруженных конструкторов, также позволяющих установить значения свойств (кроме того, в результате получается более компактный базовый программный код).

**Выполнение хранимых процедур с помощью DbCommand**

Хранимой процедурой называется блок программного кода SQL сохраненный в базе данных. Хранимые процедуры могут создаваться для того, чтобы возвращать наборы строк или скалярных типов данных, и могут иметь любое число необязательных параметров. Результатом является рабочая "единица", которая ведет себя подобно типичной функции, с той очевидной разницей, что размещается она в хранилище данных, а не в двоичном рабочем объекте.

**Замечание.**Хотя обсуждение соответствующей темы в этой главе не предполагается, самая новая версия Microsoft SQL Server (2005) включает в себя CLR-хост! Таким образом, хранимые процедуры (и другие атомарные единицы базы данных) могут создаваться с помощью управляемых языков (например, C#), а не только с помощью традиционного языка SQL. Подробности можно найти на страницах http://www.microsoft.com/sql/2005.

Для иллюстрации соответствующего процесса давайте добавим в программу CarInventoryUpdate новую опцию, которая позволит пользователю выяснить название автомобиля с помощью хранимой процедуры GetPetName. Этот объект базы данных был создан при установке базы данных Cars, и выглядит он так.

CREATE PROCEDURE **GetPetName**

@carID int,

@petName char(20) output

AS

SELECT @petName = PetName from Inventory where CarID = @carID

Сначала обновите имеющийся в Main() оператор switch, добавив в него обработку нового случая "P" для вызова новой вспомогательной функции с именем LookUpPetName(). которая принимает параметр SqlConnection и возвращает void. Обновите также метод ShowInstructions(), учитывая новый вариант выбора.

Чтобы выполнить хранимую процедуру, следует, как всегда, сначала создать новый объект соединения, сконфигурировать строку соединения и открыть сеанс. Но при создании объекта команды свойству CommandText следует присвоить имя хранимой процедуры (а не SQL-запрос). Также вы обязательно должны установить для свойства CommandType значение CommandType.StoredProcedure (значением по умолчанию является CommandType.Text).

Поскольку наша хранимая процедура имеет один входной и один выходной параметры, нашей целью является построение объекта команды, содержащего два объекта SqlParameter в своей коллекции параметров.

private static void LookUpPetName(SqlConnection cn) {

**// Получение номера машины.**

 Console.Write("Введите номер машины: ");

 int carID = int.Parse(Console.ReadLine());

**// Установка имени хранимой процедуры.**

 SqlCommand cmd = new SqlCommand("GetPetName", cn);

 cmd.CommandType = CommandType.StoredProcedure;

**// Входной параметр.**

 SqlParameter param = new SqlParameter();

 param.ParameterName = "@carID";

 param.SqlDbType = SqlDbType.Int;

 param.Value = carID;

 param.Direction = ParameterDirection.Input;

 cmd.Parameters.Add(param);

**// Выходной параметр.**

 param = new SqlParameter();

 param.ParameterName = "@petName";

 param.SqlDbType = SqlDbType.Char;

 param.Size = 20;

 param.Direction = ParameterDirection.Output();

 cmd.Parameters.Add(param);

**// Выполнение хранимой процедуры.**

 cmd.ExecuteNonQuery();

**// Печать выходного параметра.**

 Console.WriteLine("Машина {0} называется {1}'', carID, cmd.Parameters["@petName"].Value);

}

Обратите внимание на то, что свойство Direction объекта параметра позволяет указать входные и выходные параметры. По завершении вызова хранимой процедуры с помощью ExecuteNonQuery() вы можете получить значение выходного параметра, обратившись к коллекции параметров объекта команды. На рис. 22.9 показан один из возможных вариантов тестового запуска программы.



Рис. 22.9. Вызов хранимой процедуры

**Исходный код**. Проект СarsInventoryUpdater размещен в подкаталоге, соответствующем главе 22.

**Асинхронный доступ к данным в .NET 2.0**

В .NET 2.0 поставщик данных SQL (представленный пространством имен System.Data.SqlClient) усовершенствован с тем, чтобы он мог поддерживать асинхронное взаимодействие с базой данных, используя следующие новые члены SqlCommand.

• BeginExecuteReader()/EndExecuteReader()

• BeginExecuteNonQuery()/EndExecuteNonQuery()

• BeginExecuteXmlReader()/EndExecuteXmlReader()

С учетом материала, представленного в главе 14, названия пар этих методов можно считать "триумфом" соглашения о присвоении имен. Напомним, что в шаблоне асинхронного делегата .NET используется метод "begin" для выполнения задач во вторичном потоке, тогда как метод "end" может использоваться для получения результата асинхронного вызова с помощью членов IAsyncResult и необязательного делегата AsyncCallback. Поскольку работа с асинхронными командами моделируется по образцу делегата, простого примера в этом случае должно быть достаточно (но не забудьте снова заглянуть в главу 14, чтобы освежить в памяти подробности, касающиеся использования делегатов асинхронного вызова).

Предположим, что нам нужно выбрать записи из таблицы Inventory во вторичном потоке выполнения, используя объект чтения данных. Вот полный текст соответствующего метода Main() с последующим анализом.

static void Main(string[] args) {

 Console.WriteLine ("\*\*\*\*\* Забавы с ASNYC DataReader \*\*\*\*\*\n");

**// Создание открытого соединения в асинхронном режиме.**

 SqlConnection cn = new SqlConnection();

 cn.ConnectionString = "uid=sa;pwd=;Initial Catalog=Cars;" +

**"Asynchronous Processing=true;**Data Source=(local)";

 cn.Open();

**// Создание объекта SQL-команды, ожидающего около 2 с.**

 string strSQL = **"WaitFor Delay '00:00:02'**;Select \* From Inventory";

 SqlCommand myCommand = new SqlCommand(strSQL, cn);

**// Выполнение чтения во втором потоке.**

 IAsyncResult itfAsynch;

 itfAsynch = myCornmand.BeginExecuteReader(CommandBehavior.CloseConnection);

**// Выполнение действий во время работы другого потока.**

 while (!itfAsynch.IsCompleted) {

  Console.WriteLine("Работа в главном потоке…");

  Thread.Sleep(1000);

 }

 Console.WriteLine();

**// Все готово! Выполнение цикла по результатам**

**// с помощью объекта чтения данных.**

 SqlDataReader myDataReader = myCommand.EndExecuteReader(itfAsynch);

 while (myDataReader.Read()) {

  Console.WriteLine("-› Марка – {0) название – {1}, цвет – {2}.",

   myDataReader["Make"].ToString.Trim(),

   myDataReader["PetName"].ToString().Trim()*,*

   myDataReader["Color"].ToString().Trim());

 }

 myDataReader.Close();

}

Первый интересным моментом здесь является то, что вы должны разрешить асинхронное взаимодействие с помощью нового сегмента Asynchronous Processing в строке соединения. Также отметьте, что в текст объекта команды SqlCommand был добавлен сегмент WaitFor Delay для имитации длительного взаимодействия с базой данных.

Кроме этого обратите внимание на то, что вызов BeginExecuteDataReader() возвращает ожидаемый IAsyncResult-совместимый тип, который используется для синхронизации потока вызова (с помощью свойства IsCompleted), а также для получения SqlDataReader по завершении выполнения запроса.

**Исходный код**. Проект AsyncCmdObject размещен в подкаталоге, соответствующем главе22.

**Несвязный уровень ADO.NET**

Как вы убедились, работе, со связным слоем позволяет взаимодействовать с базой данных, используя объекты соединения, команд и чтения данных. С помощью небольшого набора типов вы можете по желанию выбирать, вставлять, обновлять и удалять записи (а также запускать хранимые процедуры). Но на самом деле вы пока что узнали только половину того, что вам следует знать о возможностях ADO.NET. Поэтому напоминаем, что объектная модель ADO.NET может использоваться для взаимодействия в несвязной форме.

При работе на несвязном уровне ADO.NET вы по-прежнему должны использовать объекты соединения и команды. Кроме того, вы должны использовать специальный объект, называемый *адаптером данных*(и расширяющий абстрактный DbDataAdapter), чтобы извлекать и обновлять данные. В отличие от связного слоя, данные, полученные с помощью адаптера данных, не обрабатываются с помощью объекта чтения данных. Вместо этого для перемещения данных между вызывающей стороной и источником данных объекты адаптера данных используют объекты DataSet. Тип DataSet – это контейнер, используемый для любого числа объектов DataTable, каждый из которых содержит коллекцию объектов DataRow и DataColumn.

Объект адаптера данных вашего поставщика данных обрабатывает соединение с базой данных автоматически. С целью расширения возможностей масштабируемости адаптеры данных сохраняют соединение открытым минимально возможное время. Как только вызывающая сторона получает объект DataSet, соединение с СУБД разрывается, и вызывающая сторона остается со своей локальной копией удаленных данных. Вызывающая сторона может вставлять, удалять и модифицировать данные DataTable, но физически база данных не будет обновлена до тех пор. пока вызывающая сторона не передаст явно объект DataSet адаптеру данных для обновления. В сущности, DataSet позволяет клиенту имитировать постоянно открытое соединение, в то время как реальные операции выполняются с наборами данных, находящимися в памяти (рис. 22.10).



Рис. 22.10. Объекты адаптера данных передают объекты DataSet клиенту и возвращают их обратно базе данных

Поскольку основным элементом несвязного уровня является тип DataSet, нашей следующей задачей будет выяснение того, как управлять объектом DataSet вручную. Если вы поймете, как это делается, то не будете испытывать никаких проблем При манипуляциях содержимым DataSet, извлеченным из объекта адаптера данных.

**Роль DataSet**

Упрощенно говоря, DataSet является представлением внешних данных в памяти. Более точно, DataSet представляет собой тип класса, поддерживающий три внутренние строго типизованные коллекции (рис. 22.11).



Рис. 22.11. "Анатомия" DataSet

Свойство Tables объекта DataSet позволяет получить доступ к коллекции DataTableCollection, содержащей отдельные объекты DataTable. Другой важной коллекцией DataSet является DataRelationCollection. Ввиду того, что объект DataSet является "отсоединенным" образом структуры базы данных, можно программно представлять родительски-наследственные связи между таблицами. Например, с помощью типа DataRelation можно создать отношение между двумя таблицами, моделирующее ограничение внешнего ключа, Соответствующий объект можно затем добавить в DataRelationCollection с помощью свойства Relations. После этого вы сможете осуществлять переходы между соединенными таблицами при поиске данных. Как это реализуется на практике, будет доказано немного позже.

Свойство ExtendedProperties обеспечивает доступ к объекту Property-Collection, который позволяет ассоциировать с DataSet любую дополнительную информацию, используя пары имен и значений. Эта информация может быть практически любой, и даже вообще не иметь никакого отношений к данным. Например, можно связать с DataSet название вашей компании, которое в этом случае может выступать в роли включенных в память метаданных. Другими примерами таких расширенных свойств могут быть штамп даты/времени, шифрованный пароль, который необходимо будет указать для доступа к содержимому DataSet, число, задающее частоту обновления данных, и т.д.

**Замечание**. Класс DataTable также поддерживает расширение свойств с помощью свойства ExtendedProperties.

**Члены DataSet**

Перед погружением в многочисленные детали программирования давайте рассмотрим набор базовых членов DataSet. Кроме свойств Tables, Relations и ExtendedProperties, в табл. 22.9 описаны некоторые другие интересные свойства.

**Таблица 22.9.**Свойства DataSet

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| CaseSensitive | Индикатор чувствительности к регистру cимволов при сравнении строк в объектах DataTable |
| DataSetName | Представляет понятное имя данного объекта DataSet. Обычно это значение устанавливается c помощью параметров конструктора |
| EnforceConstraints | Получает или устанавливает значение, являющееся индикатором необходимости применения заданных ограничений при любой операции обновления |
| HasErros | Получает значение, являющееся индикатором наличия ошибок в любой из строк объектов DataTable для объекта DataSet |
| RemotingFormat | Новое свойство .NET 2.0, позволяющее указать, как должна выполняться сериализация DataSet (в двоичном или XML-формате) для слоя удаленного взаимодействия .NET |

Методы DataSet воспроизводят некоторые функциональные возможности, обеспечиваемые вышеупомянутыми свойствами. В дополнение к взаимодействию с потоками XML, объект DataSet предлагает методы, позволяющие копировать/клонировать содержимое DataSet, а также устанавливать начальные и конечные точки пакетных обновлений. В табл. 22.10 даются описания некоторых из таких методов.

**Таблица 22.10**. Методы DataSet

|  |  |
| --- | --- |
| **Методы** | **Описание** |
| AcceptChanges() | Фиксирует все изменения, сделанные в данном объекте DataSet с момента его загрузки или последнего вызова AcceptChanges() |
| Clear() | Выполняет полную очистку данных DataSet путем удаления всех строк в каждом объекте DataTable |
| Clone() | Клонирует структуру DataSet, включая все объекты DataTable, а также все отношения и ограничения |
| Copy() | Копирует и структуру, и данные для имеющегося объекта DataSet |
| GetChanges() | Возвращает копию DataSet, содержащую все изменения, сделанные со времени последней загрузки или последнего вызова AcceptChanges() |
| GetChildRelations() | Возвращает коллекцию дочерних связей для указанной таблицы |
| GetParentRelations() | Возвращает коллекцию родительских связей для указанной таблицы |
| HasChanges() | Перегруженный метод, который возвращает значение, являющееся индикатором наличия модификаций у DataSet, учитывая новые, удаленные или измененные строки |
| Merge() | Перегруженный метод, который выполняет слияние данного объекта DataSet с указанным объектом DataSet |
| ReadXml() ReadXmlSchema() | Позволяют считывать XML-данные из действительного потока (файлового, размещенного в памяти или сетевого) в DataSet |
| RejectChanges() | Выполняет откат всех изменений, сделанных в DataSet с момента его создания или последнего вызова DataSet.AcceptChanges() |
| WriteXml() WriteXmlSchema() | Позволяют записать содержимое DataSet в действительный поток |

Теперь вы лучше понимаете роль DataSet (и имеете некоторое представление о том, что можно делать с этим объектом), и мы можем приступить к созданию нового консольного приложения под названием SimpleDataSet. В его методе Main() определяется новый объект DataSet, содержащий два расширенных свойства, представляющих название вашей компании и штамп времени (не забудьте указать using для System.Data).

class Program {

 static void Main(string[] args) {

  Console.WriteLine ("\*\*\*\*\* Забавы с DataSet \*\*\*\*\*\n");

**// Создание объекта DataSet.**

  DataSet carsInventoryDS = new DataSet("Inventory из Car");

  carsInventoryDS.ExtendedProperties["TimeStamp"] = DateTime.Now;

  carsInventoryDS.ExtendedProperties["Company"] = "Intertech Training";

 }

}

Объект DataSet без объектов DataTable чем-то напоминает рабочую неделю без выходных. Поэтому следующей нашей задачей будет рассмотрение внутренней структуры DataTable, начиная с типа DataColumn.

**Работа с DataColumn**

Тип DataColumn представляет отдельный столбец в пределах DataTable. Вообще говоря, набор всех типов DataColumn в границах данного типа DataTable представляет собой структуру таблицы. Например, чтобы представить таблицу Inventory базы данных Cars, вы должны создать четыре типа DataColumn, по одному для каждого столбца этой таблицы (CarID, Make, Color и PetName). После создания объектов DataColumn они добавляются в коллекцию столбцов типа DataTable (с помощью свойства Columns).

Имея определенную подготовку в области теории реляционных баз данных, вы должны знать, что столбцу в таблице данных можно назначить набор ограничений (например, использовать столбец в качестве первичного ключа, задать значение по умолчанию, потребовать, чтобы информация в столбце была доступна только для чтения и т.д.). Также каждый столбец в таблице должен соответствовать заданному для него типу данных. Например, структура таблицы Inventory требует, чтобы значения столбца CarID были целыми числами, а значения Make, Color и PetName – наборами символов. Класс DataColumn имеет множество свойств, которые позволяют выполнить соответствующие настройки. Описания основных свойств этого типа приведены в табл. 22.11.

**Таблица 22.11.**Свойства DataColumn

|  |  |
| --- | --- |
| **Свойства** | **Описание** |
| AllowDBNull | Индикатор того, что строка в этом столбце может содержать значение null. Значением по умолчанию является true |
| AutoIncrement AutoInсrementSeed AutoIncrementStep | Используются для настройки автоприращения для данного столбца, когда нужно гарантировать уникальность значений в данном объекте DataColumn (например, для первичного ключа). По умолчанию в DataColumn автоприращение не выполняется |
| Caption | Читает или устанавливает текст заголовка, который должен отображаться для данного столбца (например, текст, который увидит конечный пользователь, в DataGridView) |
| Определяет | представление DataColumn при сохранении DataSet в виде XML-документа с помощью метода DataSet.WriteXml() |
| ColumnName | Читает или устанавливает имя столбца в коллекции Columns (т.е. его представление в DataTable). Если не установить ColumnName явно, значением по умолчанию будет Column с номером столбца (т.е. Column1, Column2, Column3 и т.д.) |
| DataType | Определяет тип данных, хранимых в столбце (логический, строковый, числовой с плавающей точкой и т.д.) |
| DefaultValue | Читает или устанавливает значение, которое должно приписываться по умолчанию для данного столбца при вставке новых строк. Это значение используется тогда, когда не указано иное |
| Expression | Читает или устанавливает выражение, используемое для фильтрации строк, вычисления значений столбца или создания агрегированных столбцов |
| Ordinal | Возвращает числовую позицию столбца в коллекции Columns, поддерживаемой объектом DataTable |
| ReadOnly | Индикатор запрета изменения содержимого столбца после добавления строки в таблицу. Значением по умолчанию является false |
| Table | Возвращает объект DataTable, содержащий данный объект DataColumn |
| Unique | Индикатор требования уникальности значений в данном столбце. Если столбцу назначается ограничение первичного ключа, свойству Unique должно быть назначено значение true |

**Создание DataColumn**

Чтобы продолжить работу с проектом SimpleDataSet (и привести пример использования DataColumn), предположим, что нам нужно представить столбцы таблицы Inventory. Учитывая то, что столбец CarID является первичным ключом таблицы, мы сделаем объект DataColumn доступным только для чтения, с ограничением уникальности и не допускающим ввода значений null (используя свойства ReadOnly, Unique и AllowDBNull). Обновите метод Main() так, чтобы построить четыре объекта DataColumn.

static void Main(string[] args) {

 …

**// Создание объектов DataColumn, отображающих 'реальные'**

**// столбцы таблицы Inventory из базы данных Cars**.

 DataColumn carIDColumn = new DataColumn("CarID", typeof(int));

 carIDColumn.Caption = "Номер";

 carIDColumn.ReadOnly = true;

 carIDColumn.AllowDBNull = false;

 carIDColumn.Unique = true;

 DataColumn carMakeColumn = new DataColumn("Make", typeof(string));

 DataColumn carColorColumn = new DataColumn("Color", typeof(string));

 DataColumn carPetNameColumn = new DataColumn("PetName", typeof(string));

 carPetNameColumn.Caption = "Название";

}

**Разрешение автоприращения для полей**

Одной из возможностей DataColumn, которая может Вам понадобиться, будет возможность *автоприращения.*Автоприращение используется для того, чтобы при добавлении новой строки столбцу автоматически присваивалось значение, вычисленное на основе заданного приращения. Это может оказаться полезным тогда, когда вы хотите, чтобы столбец не имел повторяющихся значений (например, как первичный ключ).

Соответствующим поведением можно управлять с помощью свойств AutoIncrement, AutoIncrementSeed и AutoIncrementStep. Значение AutoIncrementSeed используется для начального значения столбца, а значение AutoIncrementStер задает число, которое следует добавить к AutoIncrementSeed, когда выполняется приращение. Рассмотрите следующую модификацию конструкции объекта carIDColumn типа DataColumn.

static void Main(sting[] args) {

 …

 DataColumn carIDColumn = new DataColumn("CarID", typeof(int));

 carIDColumn.ReadOnly = true;

 CarIDColumn.Caption = "Номер";

 CarIDColumn.AllowDBNull = false;

 carIDColumn.Unique = true;

 carIDColumn**.AutoIncrement**= true;

 carIDColumn**.AutoIncrementSeed** = 0;

 catIDColumn**.AutoIncrementStep =**1;

}

Здесь объект объект carIDColumn сконфигурирован так, чтобы при добавлении строк в соответствующую таблицу значение данного столбца увеличивалось на 1. Начальным значением является 0, поэтому для столбца будут выбираться числа 0, 1, 2. 3 и т.д.

**Добавление DataColumn в DataTable**

Тип DataColumn обычно не существует автономно, а добавляется в соответствующий объект DataTable.Для примера создайте новый тип DataTable (подробности будут предложены чуть позже) и вставьте объекты DataColumn в коллекцию) столбцов, используя свойство Columns.

static void Main(string[] args) {

 …

**// Добавление DataColumn в DataTable.**

 DataTable inventoryTable = new DataTable("Inventory");

 inventoryTable.Columns.AddRange(new DataColumn[] {

  carIDColumn; carMakeColumn, carColorColumn, carPetNameColumn

 });

}

**Работа с DataRow**

Вы видели, что коллекция объектов DataColumn представляет структуру DataTable. Коллекция типов DataRow представляет фактические данные таблицы. Поэтому если у вас в таблице Inventory базы данных Cars содержится 20 записей, вы можете представить эти записи с помощью 20 типов DataRow. Используя члены класса DataRow, можно вставлять, удалять оценивать и перемещать значения таблицы. Описания некоторых (но не всех) членов типа DataRow предлагаются в табл. 22.12.

**Таблица 22.12.**Основные члены типа DataRow

|  |  |
| --- | --- |
| **Члены** | **Описание** |
| HasErrors GetColumnsInError() GetColumnError() ClearErrors() RowError | Свойство HasErrors возвращает булево значение, являющееся индикатором наличия ошибок. В этом случае можно использовать метод GetColumnslnError(), чтобы получить информацию о членах, порождающих проблемы, метод GetColumnError(), чтобы получить описание ошибки, и метод ClearErrors(), удаляющий ошибки для данной строки. Свойство RowError позволяет задать текстовое описание ошибки для данной строки |
| ItemArray | Свойство, возвращающее или устанавливающее значения для данной строки с помощью массива объектов |
| RowState | Свойство, используемое для выяснения текущего "состояния" DataRow с помощью значений из перечня RowState |
| Table | Свойство, используемое для получения ссылки на DataTable, содержащий данный объект DataRow |
| AcceptChanges() | RejectChanges() Эти методы, соответственно, фиксируют или отвергают все изменения, сделанные в данной строке с момента последнего вызова AcceptChanges() |
| BeginEdit() EndEdit() CancelEdit() | Эти методы, соответственно, начинают, завершают или отменяют операции редактирования для объекта DataRow |
| Delete() | Метод, помечающий данную строку для удаления при вызове метода AcceptChanges() |
| IsNull() | Метод, возвращающий значение-индикатор того, что данный столбец содержит значение null |

Работа с DataRow немного отличается от работы с DataColumn, поскольку вы не можете создать напрямую экземпляр этого типа, а получаете ссылку от данного DataTable. Предположим, например, что нам нужно вставить две строки в таблицу Inventory. Метод DataTable.NewRow() позволяет добавить очередную строку в таблицу, а затем вы можете добавить в каждый столбец новые данные с помощью индексатора типа, как показано ниже.

static void Main(string[] args) {

 …

**// Добавление строк в таблицу Inventory.**

 DataRow carRow = inventoryTable.NewRow();

 carRow["Make"] = "BMW";

 carRow["Colar"] = "черный";

 сarRow["PetName"] = "Hamlet";

 inventoryTable.Rows.Add(carRow);

 carRow = inventoryTable.NewRow();

 carRow["Make"] = "Saab";

 carRow["Color"] = "красный";

 carRow["PetName"] = "Sea Breeze";

 inventoryTable.Rows.Add(carRow);

}

Обратите внимание на то, что класс DataRow определяет индексатор, который может использоваться для получения доступа к данному объекту DataColumn как по числовому индексу позиции, так и по имени столбца. В результате выполнения указанных строк программного кода вы получите один объект DataTable, содержащий два столбца.

**Свойство DataRow.RowState**

Свойство RowState оказывается полезным тогда, когда необходимо программно идентифицировать набор всех строк в таблице, которая, например, была изменена, только что создана и т.д. Это свойство может принимать любое значение из перечня DataRowState. Описания этих значений предлагаются в табл. 22.13.

**Таблица 22.13.**Значения перечня DataRowState

|  |  |
| --- | --- |
| **Значение** | **Описание** |
| Added | Строка была добавлена в DataRowCollection, но метод AcceptChanges() не вызывался |
| Deleted | Строка была удалена с помощью метода Delete() объекта DataRow |
| Detached | Строка была создана, но не является частью коллекции DataRowСollection. Объект DataRow находится в этом состоянии после своего создания до того, как будет добавлен к коллекции (или же после удаления этого объекта из коллекции) |
| Modified | Строка была изменена, но метод AcceptChanges() не вызывался |
| Unchanged | Строка не изменилась со времени последнего вызова AcceptChanges() |

Во время программных манипуляций строками объекта DataTable свойство RowState устанавливается автоматически.

static void Маin(string[] args) {

 …

 DataRow carRow = InventoryTable.NewRow();

**// Выводит 'Состояние строки: Detached.'**

 Console.WriteLine("Сoстояние строки: {0}.", carRow.RowState);

 carRow["Make"] = "BMW";

 carRow["Color"] = "черный";

 carRow["PetName"] = "Hamlet";

 inveritoryTable.Rows.Add(carRow);

**// Выводит 'Состояние строки: Added.'**

 Console.WriteLine("Состояние строки: {0}.",

 inventoryTable.Rows[0].RowState);

}

Как видите, DataRow в ADO.NET является достаточно "сообразительным" для того, чтобы контролировать текущее положение вещей. Поэтому, имея DataTable, вы можете выяснить, какие строки были изменены. Эта особенность DataSet очень важна, поскольку именно она при отправке обновленной информации в хранилище данных позволяет отправлять только измененные данные.

**Работа с DataTable**

Тип DataTable определяет большое количество членов, многие из которых по именам и возможностям идентичны членам DataSet. В табл. 22.14 предлагаются описания основных свойств типа DataTable, за исключением Rows и Columns.

Таблица 22.14. Основные свойства типа DataTable

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| CaseSensitive | Индикатор необходимости учета регистра символов при сравнении строк в пределах таблицы. Значением по умолчанию является false (ложь) |
| ChildRelations | Возвращает коллекцию дочерних отношений для данного объекта DataTable (если таковые имеются) |
| Constraints | Возвращает коллекцию ограничений, поддерживаемых таблицей |
| DataSet | Возвращает объект DataSet, содержащий данную таблицу (если таковой имеется) |
| DefaultView | Возвращает пользовательское представление таблицы, которое может включать фильтр или позицию курсора |
| MinimumCapacity | Читает или устанавливает значение для начального числа строк данной таблицы (это значение по умолчанию равно 25) |
| ParentRelations | Возвращает коллекцию родительских отношений для данного объекта DataTable |
| PrimaryKey | Читает или устанавливает массив столбцов, функционирующих в качестве первичных ключей для таблицы данных |
| RemotingFormat | Позволяет определить, как объект DataSet должен выполнять сериализацию соответствующего содержимого (в двоичном или XML-формате) для слоя удаленного взаимодействия .NET. Это свойство появилось в .NET 2.0 |
| TableName | Читает или устанавливает имя таблицы. Это же свойство может быть указано в качестве параметра конструктора |

В нашем примере мы установим свойство PrimaryKey типа DataTable равным объекту carIDColumn типа DataColumn.

static void Main(string[] args) {

 …

**// Установка первичного ключа для таблицы.**

 inventoryTable.PrimaryKey = new DataColumn[] { inventoryTable.Columns[0] };

}

На этом создание примера для DataTable завершается. Заключительным шагом будет вставка DataTable в DataSet-объект carsInventoryDS. Затем объект DataSet нужно оформлять вспомогательному методу PrintDataSet() (который ещё предстоит написать).

static void Main(string[] args) {

 …

**// Наконец, добавление таблицы в DataSet.**

 carsInventoryDS.Tables.Add(inventoryTable);

**// Теперь вывод данных DataSet.**

 PrintDataSet(carsInventoryDS);

}

Метод PrintDataSet() просто выполняет цикл по всем DataTable из DataSet. печатая имена столбцов и значения строк с помощью индексатора типа.

static void PrintDataSet(DataSet ds) {

 Console.WriteLine("Таблицы в DataSet '{0}'.\n", ds.DataSetName);

 foreach (DataTable dt in ds.Tables) {

  Console.WriteLine("Таблица {0}.\n", dt.TableName);

**// Вывод имен столбцов.**

  for (int curCol = 0; curCol ‹ dt.Coumns.Count; curCol++) {

   Console.Write(dt.Columns[curCol].ColumnName.Trim() + ''\t");

  }

  Console.WriteLine("\n--------------------------------");

**// Вывод DataTable.**

  for (int curRow = 0; curRow ‹ dt.Rows.Count; curRow++) {

   for (int curCol = 0; curCol ‹ dt.Columns.Count; curCol++) {

    Console.Write(dt.Rows[curRow][curCol.ToString() + "\t");

   }

   Console.WriteLine();

  }

 }

}

Вывод программы показан на рис. 22.12.



Рис. 22.12. Содержимое объекта DataSet примера

**Работа с DataTableReader в .NET 2.0**

Тип DataTable предлагает еще целый ряд методов, кроме тех, что уже были нами рассмотрены. Подобно DataSet, тип DataTable поддерживает, например, методы AcceptChanges(), GetChanges(), Сору() и ReadXml()/WriteXml(). В .NET 2.0 тип DataTable поддерживают также метод CreateDataReader(). Этот метод позволяет получить данные DataTable, используя схему, соответствующую схеме навигации объекта чтения данных (только вперед и только для чтения). Для примера создайте новую вспомогательную функцию PrintTable(), реализованную следующим образом.

private static void PrintTable(DataTable dt) {

 Console.WriteLine("\n\*\*\*\*\* Строки в DataTable \*\*\*\*\*");

**// Получение нового для .NET 2.0 типа DataTableReader.**

 DataTableReader dtReader = dt.CreateDataReader();

**// DataTableReader работает подобно DataReader.**

 while (dtReader.Read()) {

  for (int i = 0; i ‹ dtReader.FleldCount; i++) {

   Console.Write("{0} = {1} ", dtReader.GetName(i), dtReader.GetValue(i).ToString().Trim());

  }

  Console.WriteLine();

 }

 dtReader.Close();

}

Обратите внимание на то, что DataTableReader работает аналогично объекту чтения данных поставщика данных. Использование DataTableReader может оказаться идеальным вариантом, когда нужно быстро прочитать данные DataTable без просмотра внутренних коллекций строк и столбцов. Для вызова метода нужно просто указать соответствующую таблицу.

static void Main(string[] args) {

 …

**// Печать DataTable с помощью 'объекта чтения таблиц' .**

 PrintTable(carsInventoryDS.Tables["Inventory"]);

}

**Сохранение DataSet (и DataTable) в формате XML**

В завершение рассмотрения текущего примера напомним, что как DataSet, так и DataTable предлагают поддержку методов WriteXml() и ReadXml(). Метод WriteXml() позволяет сохранить содержимое объекта в локальном файле (или вообще в любом типе System.IO.Stream) в виде XML-документа. Метод ReadXml() позволяет прочитать информацию о состоянии DataSet (или DataTable) из имеющегося XML-документа. Кроме того, как DataSet, так и DataTable поддерживают WriteXmlSchema() и ReadXmlSchema() для сохранения и загрузки файлов *\**.xsd. Чтобы это проверить, добавьте в метод Main() следующий набор операторов.

static void Main(string [] args) {

 …

**// Сохранение DataSet в виде XML.**

 carsInventoryDS.WriteXml("carsDataSet.xml");

 carsInventoryDS.WriteXmlSchema("carsDataSet.xsd");

**// Очистка DataSet и вывод содержимого (должно быть пустым).**

 carsInventoryDS.Сlear();

 PrintDataSet(carsInventoryDS);

**// Загрузка и печать DataSet.**

 carsInventoryDS.ReadXml("carsDataSet.xml");

 PrintDataSet(carsInventoryDS);

}

Если открыть сохраненный файл carsDataSet.xml, вы увидите, что в нем представлены все столбцы таблицы, закодированные в виде XML-элементов.

‹?xml version="1.0" standalone="yes"?›

‹Car\_x0020\_Inventory›

 ‹Inventory›

  ‹CarID›0‹/CarID›

  ‹Make›BMW‹/Make›

  ‹Color›черный‹/Color*›*

  ‹PetName›Hamlet‹/PetName›

 ‹/Inventory›

 ‹Inventory›

  ‹CarID›1‹/CarID›

  ‹Make›Saab‹/Make›

  ‹Color›красный‹/Color›

  ‹PetName›Sea Brеeze‹/PеtName›

 ‹/Inventory›

‹/Car\_x0020\_Inventory›

Наконец, напомним, что тип DataColumn поддерживает свойство ColumnMapping, которое можно использовать для управления представлением столбца в XML-формате. Значением, устанавливаемым для этого свойства по умолчанию, является MappingType.Element. Однако можно потребовать, чтобы столбец CarID представлялся XML-атрибутом, как это сделано ниже в обновленной версии объекта carIDColumn для DataColumn.

Static void Main(string[] args) {

 …

 DataColumn carIDColumn = new DataColumn("CarID", typeof(int));

 …

 carIDColumn.ColumnMapping = MappingType.Attribute;

}

Тогда вы обнаружите следующий XML-код.

‹?xml version="1.0" standalone="yes"?›

‹Car\_x0020\_Inventory›

 ‹Inventory **CarID="0"›**

  ‹Make›BMW‹/Make›

  ‹Color›черный‹/Color›

  ‹PetName›Hamlet‹/PetName›

 ‹/Inventory›

 ‹Inventory **CarID="1"›**

‹Make›Saab‹/Make›

  ‹Color›красный‹/Color›

  ‹PetName›Sea Breeze‹/PetName›

 ‹/Inventory›

‹/Car\_x0020\_Inventory›

**Исходный код.**Проект SimpleDataSet размещен в подкаталоге, соответствующем главе 22.

**Привязка DataTable к интерфейсу пользователя**

Теперь, когда мы обсудили процесс взаимодействия с DataSets в общем, давайте рассмотрим соответствующий пример приложения Windows Forms. Нашей целью является построение формы, отображающей содержимое DataTable в рамках элемента управления DataGridView. На рис. 22.13 показано окно исходного пользовательского интерфейса проекта.



Рис. 22.13. Привязка DataTable к DataGridView

**Замечание.**Для представления реляционных баз данных в .NET 2.0 элемент управления DataGridView считается наиболее "предпочтительным", однако остается доступным и устаревший элемент управления .NET 1.x DataGrid.

Создайте новое приложение Windows Forms с именем CarDataTableViewer. Добавьте в форму элемент управления DataGridView (назвав его carInventoryGridView) и Label с подходящим описанием. Затем добавьте в проект новый C#-класс (с именем Car), определив его так, как показано ниже.

public class Car {

***//*Здесь public используется для простоты.**

 public string carPetName, carMake, carColor;

 public Car(string petName, string make, string color) {

 carPetName = petName;

 carColor = color;

**carMake = make;**

 }

}

Теперь в рамках конструктора формы, заданного по умолчанию, наполните член-переменную List‹› множеством новых объектов Car.

public partial class MainForm: System.Windows.Forms.Form {

**// Наш список машин.**

 private List‹Car› arTheCars = new List‹Car›();

 public MainForm() {

  InitializeComponent();

  CenterToScreen();

**// Заполнение списка.**

  arTheCars.Add(new Car("Chucky", "BMW", "зеленый"));

  arTheCars.Add(new Car("Tiny", "Yugo", "белый"));

  arTheCars.Add(newCar(", "Jeep", "коричневый"));

  arTheCars.Add(new Car("Pain Inducer'', "Caravan", "розовый"));

  arTheCars.Add(new Car("Fred", "BMW", "светло-зелёный"));

  arTheCars.Add(new Car ("Buddha", "BMW", "черный"));

  arTheCars.Add(new Car("Mel", "Firebird", "красный"));

  arTheCars.Add(new Car("Sarah", "Colt", "черный"));

 }

}

Как и в предыдущем примере SimpleDataSet, в приложении CarDataTableViewer будет создан объект DataTable, содержащий четыре объекта DataColumn для представления столбцов таблицы Inventory базы данных Cars. Точно так же DataTable будет содержать множество объектов DataRow для представления списка автомобилей. Но на этот раз мы заполним, строки с помощью обобщенного члена-переменной List‹›.

Во-первых, добавим в класс Form новый член-переменную с именем inventoryTable для типа DataTable. Затем добавим новую вспомогательную функцию CreateDataTable() и вызовем этот метод в конструкторе, заданном по умолчанию. Программный код, необходимый для добавлений объектов DataColumn в DataTable, идентичен программному коду предыдущего примера, поэтому здесь этот код не приводится (полный набор необходимых операторов имеется в загружаемом файле соответствующего примера). Обратите внимание на то, что здесь для построения множества строи приходится выполнить цикл по всем членам List‹›.

private void CreateDataTable() {

**// Создание объектов DataColumn и добавление их в DataTable.**

…

**// Цикл по элементам списка для создания строк.**

 foreach(Car с in arTheCars) {

  DataRow newRow = inventoryTable.NewRow();

  newRow["Make"] = c.carMake;

  newRow["Color"] = c.carColor;

  newRow["PetName"] = c.carPetName;

  inventoryTable.Rows.Add(newRow);

 }

**// Связывание DataTable с carIrventoryGridView**.

 carInventoryGridView.DataSource = inventoryTable;

}

Обратите внимание на то, что в последней строке программного кода метода CreateDataTable() свойству DataSource присваивается значение inventoryTable. Установка этого свойства и является тем единственным шагом, который необходимо выполнить для привязки DataTable к объекту DataGridView. Вы, наверное, догадываетесь, что указанный элемент графического интерфейса читает внутренние коллекции строк и столбцов. Теперь вы можете выполнить свое приложение, чтобы увидеть представление DataTable в рамках элемента управления DataGridView.

**Программное удаление строк**

Зададим себе вопрос: как удалить строку из DataTable? Одной из возможностей является вызов метода Delete() объекта DataRow, представляющего строку, которую требуется удалить. Просто укажите индекс (или объект DataRow). представляющий нужную строку. Предположим, что вы изменили графический интерфейс пользователя так, как показано на рис. 22.14.



Рис. 22.14. Удаление строк из DataTable

Следующая программная логика обработчика события Click новой кнопки обеспечивает удаление указанной строки из находящегося в памяти объекта DataTable.

**// Удаление указанной строки из DataRowCollection.**

private void btnRemoveRow\_Cl ick(object sender, EventArgs e) {

 try {

  inventoryTable.Rows[(int.Parse(txtRowToRemove.Text))].Delete();

  inventoryTable.AcceptChanges();

 } catch(Exception ex) {

  MessageBox.Show(ex.Message);

 }

}

Метод Delete(), может быть, лучше назвать MarkedAsDeletable(), поскольку строка на самом деле не будет удалена до тех пор, пока не будет вызван метод DataTable.AcceptChanges(). В действительности метод Delete() просто устанавливает для строки флаг, который сообщает от имени строки: "я готова уйти в небытие по первому же приказу моей таблицы". Также следует понимать, что даже если строка была помечена для удаления, DataTable может отменить реальное удаление с помощью RejectChanges(), как показано ниже.

**// Пометка строки для удаления с последующей отменой изменений.**

private void btnRemoveRow\_Click(object sender, EventArgs e) {

 inventoryTable.Rows[(int.Parse(txtRemove.Text))].Delete();

**// Другая работа.**

 …

 inventoryTable. RejectChanges(); // восстановление значения RowState.

}

**Применение фильтров и сортировки**

Иногда нужно показать подмножество данных DataTable, удовлетворяющее некоторому набору критериев фильтрации. Например, как показать только определенные марки автомобилей из размещенной в памяти таблицы Inventory? Метод Select() класса DataTable обеспечивает такую возможность. Обновите графический интерфейс пользователя еще раз, чтобы пользователь мог задать строку, представляющую ту марку автомобиля, которую пользователь желает видеть (рис. 22.15).

Метод Select() является перегруженным и поддерживает различную семантику выбора. Посылаемый методу Select() параметр может быть, например, строкой, содержащей соответствующие условия.



Рис. 22.15. Создание фильтра

Рассмотрите следующую программную логику обработчика события Click новой кнопки.

private void btnGetMakes\_Click(object sender, EventArgs e) {

**// Построение фильтра на основе пользовательского ввода.**

 string filterStr = string.Format("Make= '{0}' ", txtMakeToGet.Text);

**// Выбор всех строк, соответствующих фильтру.**

 DataRow[] makes = inventoryTable.Select(filterStr);

**// Вывод того, что получилось.**

 if (makes.Length == 0) MessageBox.Show("Извините, машин нет…", "Ошибка выбора!");

 else {

  string strMake = null;

  for (int i = 0; i ‹ makes.Length; i++) {

   DataRow temp = makes[i];

   strMake += temp["PetName"] + "\n";

  }

  MessageBox.Show(strMake, txtMakeToGet.Text + " type(s):");

 }

}

Здесь строится простой фильтр на основе значения, содержащегося в соответствующем TextBox. Если вы укажете BMW, то фильтр будет иметь вид Make=BMW. Отправив этот фильтр методу Select(), вы получите массив типов DataRow. представляющих те строки, которые соответствуют данному фильтру (рис. 22.16).



Рис. 22.16. Отображение отфильтрованных данных

Как видите, программная логика фильтра имеет стандартный синтаксис SQL, Чтобы проверить это, предположим, что требуется представить результаты предыдущего вызова Select() в алфавитном порядке. В терминах SQL это означает сортировку до столбцу PetName. Поскольку метод Select() является перегруженным. мы можем указать критерий сортировки так, как показано ниже.

**// Сортировка по PetName.**

makes = inventoryTable.Select(filterStr, "**PetName**");

Чтобы увидеть результаты в нисходящем порядке, вызовите Select(), как показано ниже.

**// Возвращает результаты в порядке убывания.**

makes = inventoryTable.Select(filterStr*,*"**PetName DESC**");

Вообще говоря, строка сортировки должна содержать имя столбца, за которым следует "ASC" (обозначающее порядок по возрастанию, что является значением, принимаемым по умолчанию) или "DESC" (обозначающее порядок по убыванию). При необходимости можно указать несколько столбцов, разделив их запятыми. Наконец, следует понимать, что в строке фильтра можно использовать любое число операций отношения. Например, если требуется найти все автомобили с номерами, большими 5, то вот как должна выглядеть вспомогательная функция, которая сделает это.

private void ShowCarsWithIdLessThanFive() {

**// Вывод названий машин с номерами, большими 5.**

 DataRow[] properIDs;

 string newFilterStr = "ID › 5";

 properIDs = inventoryTable.Select(newFilterStr);

 string strIDs = null;

 for (int i = 0; i ‹ properIDs.Length; i++) {

  DataRow temp = properIDs[i];

  strIDs += temp["PetName"] + " is ID " + temp["ID"] + "\n";

 }

 MessageBox.Show(strIDs, "Названий машин с ID › 5");

}

**Обновление строк**

Еще одной операцией, которую вы должны освоить, является изменение значений существующей в DataTable строки. С этой целью можно, например, сначала c помощью метода Select() получить строку, соответствующую имеющемуся критерию фильтра. Имея соответствующий объект DataRow, вы можете соответствующим образом его изменить. Предположим, что в форме есть кнопка (тип Button). при щелчке на которой выполняется поиск тех строк в объекте DataTable, для которых Make равно BMW. Идентифицировав эти элементы, вы изменяете значение Make с BMW на Colt.

**// Поиск строк для редактирования с помощью фильтра.**

private void btnChangeBeemersToColts\_Click(object sender, EventArgs e) {

**// Проверка вменяемости пользователя.**

 if (DialogResult.Yes == MessageBox.Show("Вы уверены?? BMW намного лучше, чем Colt!", "Подтвердите свой выбор!", MessageBoxButtons.YesNo)) {

**// Построение фильтра.**

  string filterStr = "Make='BMW'";

  string strMake = null;

**// Поиск строк, соответствующих критериям фильтра.**

  DataRow[] makes = inventoryTable.Select(filterStr);

**// Замена бумеров на кольты!**

  for (int i = 0; i ‹ makes.Length; i++) {

   DataRow temp = makes[i];

   strMake += temp["Make"] = "Colt";

   makes[i] = temp;

  }

 }

}

Класс DataRow предлагает методы BeginEdit(), EndEdit() и CancelEdit(), которые позволяют редактировать содержимое строки, временно приостанавливая все заданные условия проверки ввода. При использовании предложенной выше программной логики строка будет проверяться с каждым новым присваиванием. (И если вы задали обработку каких-то событий DataRow, они тоже будут генерироваться с каждой модификацией.) При вызове BeginEdit() для DataRow строка переводится в режим редактирования. Вы получаете возможность внести любые необходимые вам изменения, чтобы затем вызывать EndEdit() и зафиксировать эти изменения или CancelEdit() и выполнить откат к оригинальной версии данных, например:

private void UpdateSomeRow() {

**// Предполагается, что строка для редактирования уже получена.**

**// Выполняется перевод этой строки в режим редактирования.**

 rowToUpdate.BeginEdit();

**// Отправка строки вспомогательной функции, возвращающей Boolean.**

 if (ChangeValuesForThisRow(rowToUpdate)) rowToUpdate.EndEdit(); **// OK!**

 else rowTaUpdate.CancelEdit(); **// Забудьте об этом.**

}

Вы, конечно, можете вызывать эти методы для данного DataRow и вручную, но они вызываются автоматически при редактировании элемента DataGridView, связанного с DataTable. Например, при выборе строки в DataGridView эта строка автоматически переводится в режим редактирования. А при перемещении фокуса ввода в новую строку автоматически вызывается EndEdit().

**Работа с типом DataView**

В терминах базы данных *представление -* это показ набора данных таблицы (или множества таблиц) в определенном стиле. Например, с помощью Microsoft SQL Server на базе таблицы Inventory можно создать представление, которое возвратит новую таблицу, содержащую автомобили только заданного цвета. В ADO.NET тип DataView позволяет программно извлечь подмножество данных из DataTable и разместить их в отдельном объекте. Одним из важных преимуществ создания множества представлений одной и той же таблицы: является то, что вы можете связать эти представления с различными элементами графического интерфейса (такими, как DataGridView). К примеру, один элемент DataGridView можно связать с объектом DataView, показывающим все машины из таблицы Inventory, в то время как другой элемент будет настроен на отображение только зеленых автомобилей.

Для иллюстрации добавите к текущему графическому интерфейсу еще один тип DataGridView, назвав его dataGridColtsView и сопроводив поясняющим элементом Label. Затем определите член-переменную coltsOnlyView типа DataView.

public partial class MainForm: Form {

**// Представление для DataTable.**

 DataView coltsOnlyView;

**// Отображение только красных кольтов.**

 …

}

Теперь создайте новую вспомогательную функцию с именем CreateDataView() и вызовите этот метод в конструкторе формы, заданном по умолчанию, сразу же после того, как будет создан тип DataTable:

public MainForm() {

 …

**// Создание таблицы данных.**

 CreateDataTable();

**// Создание представления.**

 CreateDataView();

}

Ниже показана реализация этой новой вспомогательной функции. Обратите внимание на то, что конструктору DataView передается тип DataTable, который будет использоваться для построения пользовательского набора cтрок данных.

private void CreateDataView() {

**// Установка таблицы для представления.**

 coltsOnlyView = new DataView(inventoryTable);

**// Настройка представления с помощью фильтра.**

 coltsOnlyView.RowFilter = "Make = 'Colt'";

**// Привязка к элементу управления,**

 dataGridColtsView.DataSource = coltsOnlyView;

}

Как видите, класс DataView предлагает свойство RowFilter, содержащее строку критериев фильтрации, используемую для поиска соответствующих строк. После получения объекта представления соответственно устанавливается свойство DataSource элемента управления. На рис. 22.17 показано окно готового приложения.

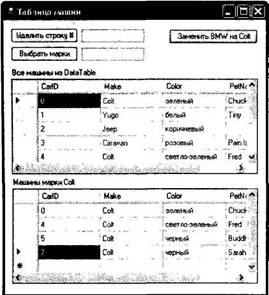


Рис. 22.17. Представление отфильтрованных данных

**Исходный код.**Проект CarDafaTableViewer размещен в подкаталоге, соответствующем главе 22.

**Работа с адаптерами данных**

Теперь, когда вы знаете возможности использования типов DataSet ADO.NET, обратим внимание на адаптеры данных. Напомним, что объекты адаптера данных используются для "наполнения" DataSet объектами DataTable и возврата измененных объектов DataTable базе данных для обработки. Описания основных членов базового класса DbDataAdapter приведены в табл. 22.15.

**Таблица 22.15. Основные члены класса**DbDataAdapter

|  |  |
| --- | --- |
| **Члены** | **Описание** |
| SelectCommand InsertCommand UpdateCommand DeleteCommand | Задают SQL-команды, которые будут отправлены хранилищу данных при вызове метода Fill() или Update() |
| Fill() | Заполняет данную таблицу в DataSet некоторым набором записей, зависящим от заданного объектом команды значения SelectCommand |
| Update() | Обновляет DataTable, используя объекты команд из свойств InsertCommand, UpdateCommand или DeleteCommand. Точная команда, которая при этом выполняется, зависит от значения RowState для данного DataRow в данном объекте DataTable (данного DataSet) |

В следующих примерах не забывайте о том, что объекты адаптера данных управляют соответствующим соединением с базой данных за вас, так что вам не придется явно открывать или закрывать сеанс связи с СУБД. Тем не менее, вам все равно нужно предоставить адаптеру данных действительный объект соединения или, в виде аргумента конструктора, строку соединения (которая будет использоваться для построения внутреннего объекта соединения).

**Заполнение DataSet с помощью адаптера данных**

Создайте новое консольное приложение с именем FillDataSetWithSqlDataAdapter, указав в нем использование пространств имен System.Data и System. Data.SqlClient. Обновите метод Main() так, как предлагается нише (для простоты здесь не показан блок try/catch).

static void Main(string[] args) {

 Console.WriteLine("\*\*\*\*\* Забавы с адаптерами данных \*\*\*\*\* \n");

 string cnStr = "uid=sa;pwd=;Initial Catalog=Cars;Data Source=(local)";

**// Заполнение DataSet новыми DataTable.**

 DataSet myDS = new DataSet("Cars");

 SqlDataAdapter dAdapt *=*new SqlDataAdapter("Select \* From Inventory".cnStr);

 dAdapt.Fill(myDS, "Inventory");

**// Отображение содержимого.**

 PrintDataSet(myDS)**;**

}

Обратите внимание на то, что адаптер данных создается с указанием SQL-оператора Select. Это значение будет использоваться для внутреннего построения объекта команды, которую затем можно будет получить, выбрав свойство SelectCommand. Далее, заметьте, что метод Fill() получает экземпляр типа DataSet и необязательное строковое имя, которое будет использоваться при установке свойства TableName нового объекта DataTablе (если вы не укажете имя таблицы, адаптер данных использует для таблицы имя Table).

**Замечание**. Метод Fill() возвращает целое число, соответствующее числу строк, затронутых SQL-запросом.

Как и следует ожидать, при передаче DataSet методу PrintDataSet() (реализованному в этой главе ранее) будет получен список всех строк таблицы Inventory базы данных Cars (рис. 22.18).

**Отображение имен базы данных в понятные имена**

Вы, скорее всего, знаете, что администраторы баз данных склонны создавать имена, таблиц и столбцов, которые нельзя назвать понятными для конечных пользователей. Но хорошей вестью является то, что объекты адаптера данных поддерживают внутреннюю строго типизованную коллекцию (DatаTableМаррing-Collection) типов System.Data.Common.DataTableMapping, доступную с помощью свойства TableMappings.



Рис. 22.18. Заполнение DataSet с помощью объекта адаптера данных

При желании вы можете использовать эту коллекцию для того, чтобы информировать DataTable о "дисплейных именах", которые должны использоваться при выводе содержимого. Предположим, например, что вы хотите отобразить имя Inventory, используемое для таблицы в рамках СУБД, в дисплейное имя Ассортимент. Кроме того, предположим, что вы хотите отобразить имя столбца CarID в виде Номер, а имя столбца PetName – в виде Название. Для этого в объект адаптера данных перед вызовом метода Fill() добавьте следующий программный код (и не забудьте указать using для пространства имен System.Data.Common).

static void Main(string[] args) {

 …

**// Отображение имен столбцов БД в имена, понятные пользователю.**

 DataTableMapping custMap = dAdapt.TableMappings.Add("Inventory", "Ассортимент");

 custMap.ColumnMappings.Add("CarID", "Номер");

 custMap.ColumnMappings.Add("PetName", "Название");

 dAdapt.Fill(myDS, "Inventory");

 …

}

Теперь, выполнив программу, вы обнаружите, что метод PrintDataSet() отображает "понятные" имена объектов DataTable и DataRow, а не имена, заданные структурой базы данных.

**Исходный код.**Проект FillDataSetWithSqIDataAdapter размещен в подкаталоге, соответствующем главе 22.

**Обновление базы данных с помощью объекта адаптера данных**

Адаптеры данных могут не только заполнять для вас таблицы объекта DataSet. Они могут также поддерживать набор объектов основных SQL-команд, используя их для возвращения модифицированных данных обратно в хранилище данных. При вызове метода Update() адаптера данных проверяется свойство RowState для каждой строки в DataTable и используются соответствующие SQL-команды, присвоенные свойствам DeleteCommand, InsertCommand и UpdateCommand, чтобы записать изменения данного DataTable в источник данных.

Чтобы проиллюстрировать процесс использования адаптера данных для возвращения изменении DataTable в хранилище данных, в следующем примере мы переработаем приложение CarsInventoryUpdater, созданное в этой главе ранее, чтобы на этот раз использовать DataSet и объект адаптера данных. Поскольку значительная часть приложения останется той же, сконцентрируем свое внимание на изменениях, которые необходимо сделать в методах DeleteCar(). UpdateCarPetName() и InsertNewCar() (чтобы уточнить детали, проверьте текст загружаемого программного кода для данного примера).

Первым основным изменением, которое требуется внести в приложение, является определение двух новых статических членов-переменных класса Program для представления DataSet и объекта соединения. Также, чтобы заполнить DataSet начальными данными, модифицируется метод Main().

class Program {

**// Объект DataSet, доступный на уровне приложения.**

 public static DataSet dsСarInventory = new DataSet("CarsDatabase");

**// Объект соединения, доступный на уровне приложения.**

 public static SqlConnection cnObj = new SqlConnection("uid-sa;pwd=;Initial Catalog=Cars;Data Source= (local)");

 static void Main(string[] args) {

**// Создание адаптера данных и заполнение DataSet.**

  SqlDataAdapter dAdapter = new SqlDataAdapter("Select \* From Inventory", chObj);

  dAdapter.Fill(dsCarInventory, "Inventory");

  ShowInstructions();

**// Программный код получения команды пользователя…**

 }

 …

}

Обратите внимание и на то, что методы ListInventory(), DeleteCar(), UpdateCarPetName() и InsertNewCar() также были изменены с тем, чтобы они могли принять SqlDataAdapter в качестве параметра.

**Установка свойства InsertCommand**

При использовании адаптера данных для обновления DataSet первой задачей оказывается назначение свойствам UpdateCommand, DeleteCommand и InsertCommand действительных объектов команд (пока вы этого не сделаете, эти свойства возвращают null). Слово "действительные" для объектов команд здесь используется потому, что набор объектов команд, которые вы "подключаете" к адаптеру данных, изменяется в зависимости от таблицы, данные которой вы пытаетесь обновить. В этом примере соответствующей таблицей является таблица Inventory. И вот как выглядит измененный метод InsertNewCar().

private static void InsertNewCar(SqlDataAdapter dAdapter) {

**// Сбор информации о новой машине.**

 …

**// Формирование SQL-оператора Insert и подключение к DataAdapter.**

 string sql = string.Format("Insert Into Inventory" +

  "(CarID, Make, Color, PetName) Values" +

  "('{0}', '{1}', '{2}', '{3}')",

  newCarID, newCarMake, newCarColor, newCarPetName);

 dAdapter.InsertCommand = new SqlCommand(sql);

 dAdapter.InsertCommand.Connection = cnObj;

**// Обновление таблицы Inventory с учетом новой строки.**

 DataRow newCar = dsCarInventory.Tables["Inventory"].NewRow();

 newCar["CarID"] = newCarID;

 newCar["Make"] = newCarMake;

 newCar["Color"] = newCarColor;

 newCar["PetName"] = newCarPetName;

 dsCarInventory.Tables["Inventory"].Rows.Add(newCar);

 dAdapter.Update(dsCarInventory.Tables["Inventory"]);

}

После создания объекта команды он "подключается" к адаптеру с помощью свойства InsertCommand. Затем в DataTable таблицы Inventory добавляется новая строка, представленная объектом dsCarInventory. После добавления DataRow в DataTable адаптер выполнит SQL-команду, обнаруженную в свойстве InsertCommand, поскольку значением RowState этой новой строки является DataRowState.Added.

**Установка свойства UpdateCommand**

Модификации метода UpdateCarPetName() оказываются приблизительно такими же. Просто создайте новый объект команды и укажите его для свойства UpdateCommand.

private static void UpdateCarPetName(SqlDataAdapter dAdapter) {

**// Сбор информации об изменяемой машине.**

 …

**// Форматирование SQL-оператора Update и подключение к DataAdapter.**

 string sql = string.Format("Update Inventory Set PetName = '{0}' Where CarID = '{1}'", newPetName, carToUpdate);

 SqlCommand cmd = new SqlCommand(sql, cnObj);

 dAdapter.UpdateCommand = cmd;

 DataRow[] carRowToUpdate = dsCarInventory.Tables["Inventory"].Select(string.Format("CarID = '{0}'", carToUpdata));

 carRowToUpdate[0]["PetName"] = newPetName;

 dAdapter.Update(daCarInventory.Tables["Inventory"]);

}

В данном случае, когда вы выбираете строку (с помощью метода Select()), для RowState указанной строки автоматически устанавливается значение DataRowState.Modified. Другим заслуживающим внимания моментом здесь является то, что метод Select() возвращает массив объектов DataRow, поэтому вы должны указать, какую именно строку требуется изменить.

**Установка свойства DeleteCommand**

Наконец, вы имеете следующую модификацию метода DeleteCar().

private static void DeleteCar(SqlDataAdaper dAdapter) {

**// Получение номера удаляемой машины.**

 …

 string sql = String.Format("Delete from Inventory where CarID = '{0}'"; carToDelete);

 SqlCommand cmd = new SqlCommand(sql, cnObj);

 dAdapter.DeleteCommand = cmd;

 DataRow[] carRowToDelete = dsCarInventory.Tables["Inventory"].Select(string.Format("CarID = '{0}'", carToDelete));

 carRowToDelete[0].Delete();

 dAdapter.Update(dsCarInventory.Tables["Inventory"]);

}

В этом случае вы находите строку, которую нужно удалить (снова с помощью метода Select()), а затем устанавливаете для свойства RowState значение DataRowState.Deleted с помощью вызова Delete().

**Исходный код.**Проект CarslnventoryUpdaterDS размещен в подкаталоге, соответствующем главе 22.

**Генерирование SQL-команд с помощью типов построителя команд**

Вы должны согласиться с тем, что для работы с адаптерами данных может потребоваться ввод довольно большого объема программного кода, а также создание всех четырех объектов команд и соответствующей строки соединения (или DbConnection-объекта). Чтобы упростить дело, в .NET 2.0 каждый из поставщиков данных ADO.NET предлагает тип *построителя команд.*Используя этот тип, вы можете автоматически получать объекты команд, содержащие правильные типы команд Insert, Delete и Update на базе исходного оператора Select.

Тип SqlCommandBuilder автоматически генерирует значения для свойств InsertCommand, UpdateCommand и DeleteCommand объекта SqlDataAdapter на основе значения SelectCommand. Очевидным преимуществом здесь является то, что исключается необходимость строить все типы SqlCommand и SqlParameter вручную.

Здесь возникает вопрос о том, как построитель команд может строить указанные объекты SQL-команд "на лету". Оказывается, все дело в метаданных. В среде выполнения, когда вы вызываете метод Update() адаптера данных, соответствующий построитель команд читает данные структуры базы данных для автоматического генерирования объектов соответствующих команд вставки, удаления и обновления данных.

Рассмотрите следующий пример, в котором строка из DataSet удаляется с помощью автоматически сгенерированных SQL-операторов. Кроме того, для каждого объекта команды соответствующая команда выводится на печать.

static void Main(string[] args) {

 DataSet theCarsInventory = new DataSet();

**// Создание соединения.**

 SqlConnection cn = new SqlConnection("server=(local);User ID=sa;Pwd=;database=Cars");

**// Автоматическое генерирование команд Insert, Update и Delete**

**// на основе существующей команды Select.**

 SqlDataAdapter da = new SqlDataAdapter("SELECT \* FROM Inventory", cn);

 SqlCommandBuilder invBuilder = new SqlCommandBuilder(da);

**// Заполнение DataSet.**

 da.Fill(theCarsInventory, "Inventory");

 PrintDataSet(theCarsInventory);

**// Удаление строки на основании пользовательского ввода**

**// и обновление базы данных.**

 try {

  Console.Write("Номер строки для удаления: ");

  int rowToDelete = int.Parse(Console.ReadLine());

  theCarsInventory.Tables["Inventory"].Rows[rowToDelete].Delete();

  da.Update(theCarsInventory, "Inventory");

 } catch (Exception e) {

  Console.WriteLine(e.Message);

 }

**// Новое заполнение и печать таблицы Inventory.**

 theCarsInventory = new DataSet();

 da.Fill(theCarsInventory, "Inventory");

 PrintDataSet(theCarsInventory);

}

В этим фрагменте программного кода обратите внимание на то, что здесь объект построителя команд (в данном случае это SqlCommandBuilder) передается объекту адаптера данных в виде параметра конструктора и больше никак не используется. Как бы странно это ни выглядело, но это все, что вам требуется сделать (в минимальных условиях). Указанный тип в фоновом режиме сконфигурирует для адаптера данных остальные объекты команд.

Хотя идея получить кое-что, не предлагая ничего взамен, понравится многим, важно понять, что построители команд имеют некоторые (весьма важные) ограничения. В частности, построитель команд может автоматически генерировать SQL-команды для адаптера данных только в том случае, когда выполнены все следующие условия.

• Соответствующая команда Select взаимодействует только с одной таблицей (так что, например, соединения таблиц не допускаются).

• Для этой единственной таблицы назначен первичный ключ.

• Столбцы, представляющие первичный ключ, присутствуют в данном SQL-операторе Select.

Так или иначе, рис. 22.19 демонстрирует, что указанная строка удаляется из физической базы данных (при анализе программного кода этого примера следует различать значение CarID и порядковый номер строки).



Рис. 22.19. Использование автоматически генерируемых команд SQL

**Исходный код**. Проект MySqlCommandBuilder размещен в подкаталоге, соответствующем главе 22.

**Объекты DataSet с множеством таблиц и объекты DataRelation**

До этого момента во всех примерах данной главы объекты DataSet содержали по одному объекту DataTable. Однако вся мощь несвязного уровня ADO.NET проявляется тогда, когда DataSet содержит множество объектов DataTable. В этом случае вы можете добавить в коллекцию DataRelation объекта DataSet любое число объектов DataRelation, чтобы указать взаимные связи таблиц. Используя эти объекты, клиентское звено приложения сможет осуществлять переходы между таблицами без пересылки данных по сети.

Для демонстрации возможностей использования объектов DataRelation создайте новый проект Windows Forms с именем MultitabledDataSet. Графический интерфейс пользователя этого приложения достаточно прост. На рис. 22.20 вы можете видеть три элемента управления DataGridView, содержащие данные из таблиц Inventory, Orders и Customers базы данных Cars. Кроме того, там присутствует одна кнопка, с помощью которой информация обо всех изменениях направляется в хранилище данных.

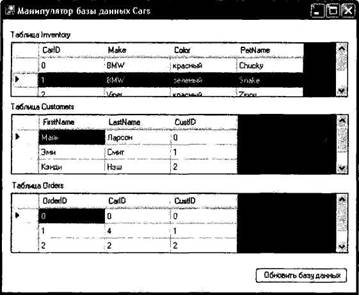


Рис. 22.20. Просмотр связанных объектов DataTable

Чтобы упростить ситуацию, тип MainForm будет использовать построители команд (по одному для каждой таблицы) для автоматического генерирования SQL-команд каждого из трех объектов SqlDataAdapter. Вот исходная модификация соответствующего экземпляра типа Form:

public partial class MainForm: Form {

**// Объект DataSet для формы.**

 private DataSet carsDS = new DataSet("CarsDataSet");

**// Применение построителей команд для упрощения**

**// настройки адаптеров данных.**

 private SqlCommandBuilder sqlCBInventory;

 private SqlCommandBuilder sqlCBCustomers;

 private SqlCommandBuilder sqlCBOrders;

**// Адаптеры данных (для каждой из таблиц).**

 private SqlDataAdapter intTableAdapter;

 private SqlDataAdapter custTableAdapter;

 private SqlDataAdapter ordersTableAdapter;

**// Объект соединения для формы.**

 private SqlConnection cn = new SqlConnection("server= (local);uid=sa;pwd=;database=Cars");

 …

}

Конструктор формы выполняет основную работу по созданию членов-переменных для данных и заполнению DataSet. Обратите также внимание на вызов приватной вспомогательной функции ВuildTableRelationship().

public MainForm() {

 InitializeComponent();

**// Создание адаптеров.**

 invTableAdapter = new SqlDataAdapter("Select \* from Inventory", cn);

 custTableAdapter = new SqlDataAdapter("Select \* from Customers", cn);

 ordersTableAdapter = new SqlDataAdapter("Select \* from Orders", cn);

**// Автогенерирование команд.**

 sqlCBInventory = new SqlCommandBuilder(invTableAdapter);

 sqlCBOrders = new SqlCommandBuilder(ordersTableAdapter);

 sqlCBCustomers = new SqlCommandBuilder(custTableAdapter);

**// Добавление таблиц в DataSet.**

 invTableAdapter.Fill(carsDS, " Inventory");

 custTableAdapter.Fill(carsDS, "Customers");

 ordersTableAdapter.Fill(carsDS, "Orders");

**// Создание отношений между таблицами.**

 BuildTableRalationship();

**// Привязка к элементам управления.**

 dataGridViewInventory.DataSource = carsDS.Tables["Inventory"];

 dataGridViewCustomers.DataSourсе = carsDS.Tables["Customers"];

 dataGridViewOrders.DataSource = carsDS.Tables["Orders"];

}

Вспомогательная функция BuildTableRelationship() делает в точности то, что от нее ожидается. Напомним, что база данных Cars имеет ряд отношений "родитель-потомок", что учитывается в следующем фрагменте программного кода:

private void BuildTableRelationship() {

**// Создание объекта отношения CustomerOrder.**

 DataRelation dr = new DataRelation("CustomerOrder", carsDS.Tables["Customers"].Columns["CustID"], carsDS.Tables["Orders"].Columns["CustID"]);

 carsDS.Relations.Add(dr);

**// Создание объекта отношения InventoryOrder.**

 dr = new DataRelation("InventoryOrder", carsDS.Tables["Inventory"].Columns["CarID"], carsDS.Tables["Orders"].Columns["CarID"]);

 carsDS.Relations.Add(dr);

}

Теперь, когда объект DataSet заполнен и отсоединен от источника данных, вы можете работать со всеми таблицами локально. Можно вставлять, обновлять или удалять значения любого из трех DataGridView. Как только вы будете готовы отправить измененные данные обратно в хранилище данных для обработки, щелкните на кнопке обновления формы. Программный код соответствующего обработчика события Click должен быть вам понятен.

private void btnOpdate\_Cliсk(object sender, EventArgs e) {

 try {

  invTableAdapter.Update(carsDS, "Inventory");

  custTableAdapter.Update(carsDS, "Customers");

  ordersTableAdapter.Update(carsDS, "Orders");

 } catch (Exception ex) {

  MessageBox.Show(ex.Message);

 }

}

После щелчка на кнопке обновления вы обнаружите, что каждая из таблиц в базе данных Cars соответствующим образом изменена.

**Навигационные возможности для связанных таблиц**

Чтобы продемонстрировать возможности DataRelation при программной реализации доступа к данным связанных таблиц, добавьте в форму новый тип Button и соответствующий ему TextBox. В результате конечный пользователь должен получить возможность ввести идентификационный номер заказчика и увидеть информацию о заказе соответствующего клиента, которая выводится в простом окне сообщения. Обработчик события Click этой кнопки реализован так.

private void btnGetInfo\_Click(object sender, System.EventArgs e) {

 string strInfo = "";

 DataRow drCust = null;

 DataRow[] drsOrder = null;

**// Получение указанного CustID из TextBox.**

 int theCust = int.Parse(this.txtCustID.Text);

**// Получение для CustID соответствующей строки таблицы Customers.**

 drCust = carsDS.Tables["Customers"].Row[theCust];

 strInfo += "Заказчик №" + drCust["CustID"].ToString() + "\n";

**// Переход от таблицы заказчиков к таблице заказов.**

 drsOrder = drCust.GetChildRows(carsDS.Relations["CustomerOrder"]);

**// Получение номера заказа.**

 foreach (DataRow r in drsOrder) strInfo += "Номер заказа: " + r["OrderID"] + "\n";

**// Переход от таблицы заказов к таблице ассортимента.**

 DataRow[] drsInv = drsOrder[0].GetParentRows(carsDS.Relatios["InventoryOrder"]);

**// Получение информации о машине.**

 foreach (DataRow r in drsInv) {

  strInfo += "Марка: " + r["Make"] + "\n";

  strInfo += "Цвет: " + r["Color"] + "\n";

  strInfo += "Название: " + r["PetName"] + "\n";

 }

 MessageBox.Show(stxInfo, "Информация для данного заказчика");

}

Как видите, ключом к решению задачи перемещения между таблицами данных оказывается использование ряда методов, определённых типом DataRow. Давайте разберем этот программный код но порядку. Сначала вы получаете подходящий идентификационный номер заказчика из текстового блока и используете этот номер для того, чтобы найти соответствующую строку в таблице Customers (конечно же, с помощью свойства Rows), как показано ниже.

**// Получение указанного CustID из TextBox.**

int theCust = int.Parse(this.txtCustID.Text);

**// Получение для CustID соответствующей строки таблицы Customers.**

DataRow drCust = null;

drCust = carsDS.Tables["Customers"].Raws[theCust];

strInfo += "3аказчик №" + drCust["CustID"].ToString() *+*"\n";

Затем вы переходите от таблицы Customers к таблице Orders, используя отношение CustomerOrder. Обратите внимание на то, что метод DataRow.GetChildRows() позволяет получить доступ к строкам дочерней таблицы. После этого вы можете прочитать информацию из этой таблицы.

**// Переход от таблицы заказчиков к таблице заказов.**

DataRow[] drsOrder = null;

drsOrder = drCast.**GetChildRows**(carsDS.Relations["CustomerOrder"]);

**// Получение номера заказа.**

foreach(DataRow r in drsOrder) strInfo += "Номер заказа: " + r["OrderID"] + "\n";

Заключительным шагом является переход от таблицы Orders к родительской таблице (Inventory) с помощью метода GetParentRows(). После этого вы сможете прочитать информацию из таблицы Inventory для столбцов Make*,*PetName и Color. как показано ниже.

**// Переход от таблицы заказов к таблице ассортимента.**

DataRow[] drsInv = drsOrder[0].GetParentRows(carsDS.Relations["InventoryOrder"]);

foreach(DataRow r in drsInv) {

 strInfo += "Марка: " + r["Make"] + "\n";

 strInfo += "Цвет: " + r["Color"] + "\n";

 strInfo += "Название: " + r["PetName"] + "\n";

}

На рис. 22.21 показан один из возможных вариантов вывода.



Рис. 22.21. Навигация по связанным данным

Этот пример убеждает в пользе типа DataSet. Поскольку DataSet отсоединяется от соответствующего источника данных, вы можете работать с копией данных, размещенной в памяти, переходя от одной таблицы к другой и выполняя все необходимые модификации, удаления или вставки. По завершении этой работы вы можете направить свои изменения в хранилище данных для их обработки.

**Исходный код.**Проект MultitabledDataSetApp размещен в подкаталоге, соответствующем главе 22.

**Возможности мастеров данных**

К этому моменту нашего рассмотрения вы открыли для себя множество путей взаимодействия с типами ADO.NET без использования мастеров. Но, хотя понимание основ работы с поставщиком данных оказывается (определенно) важным, важно и то, что от больших объемов вводимого вручную, в общем-то, шаблонного программного кода могут болеть руки. Поэтому в завершение мы рассмотрим несколько мастеров данных, которые могут вам при случае пригодиться.

Здесь не ставится цель комментировать все мастера данных для интерфейса пользователя, имеющиеся в Visual Studio 2005, но чтобы показать их основные возможности, мы рассмотрим некоторые дополнительные опции конфигурации элемента управления DataGridView. Создайте новое приложение Windows Forms с одной формой, содержащей элемент управления DataGridView с именем inventoryDataGridView. В окне проектирования формы активизируйте встроенный редактор этого элемента, и в раскрывающемся списке Choose Data Source (Выбрать источник данных) щелкните на ссылке Add Project Data Source (Добавить источник данных в проект), рис. 22.22.

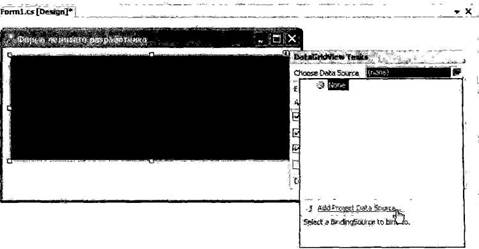


Рис. 22.22. Добавление источника данных

В результате будет запущен мастер конфигураций источников данных. На первом шаге проста выберите пиктограмму Database и щелкните на кнопке Next. На втором шаге щелкните на кнопке New Connection (Новое соединение) и установите связь с базой данных Cars (используя вышеприведенные инструкции из раздела "Соединение с базой данных в Visual Studio 2005" этой главы). Третий шаг позволяет мастеру сохранить строку соединения во внешнем файле Арр.config в рамках должным образом сконфигурированного элемента ‹connectionStrings› (что, в общем-то, является довольно хорошим решением). На заключительном шаге вы получаете возможность выбрать объекты базы данных, которые должны использоваться генерируемым объектом DataSet, и для вашего примера это будет одна таблица Inventory (рис. 22.23).

По завершении работы мастера вы увидите, что элемент DataGridView в окне проектирования формы автоматически отображает имена столбцов. И если выполнить приложение в том виде, в каком оно находится сейчас, вы увидите все содержимое таблицы Inventory, отображенное в окне указанного элемента интерфейса. Если рассмотреть программный код для события Load вашей формы, вы обнаружите, что элемент управлений заполняется с помощью строки программного кода, выделенной здесь полужирным шрифтом.

public partial class MainForm: Form {

 public MainForm() {

  InitializeComponent();

 }

 private void MainForm\_Load(object sender, EventArgs e) {

  // TODO: This line of code loads data into

  // the 'carsDataSet.Inventory' table.

  // You can move, or remove it, as needed.

**this.inventoryTableAdapter.Fill(this.carsDataSet.Inventory);**

 }

}

Чтобы понять, что делает эта строка программного кода, мы должны сначала выяснить роль строго типизованных объектов DataSet.

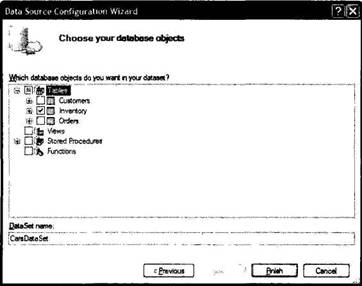


Рис. 22.23. Выбор таблицы Inventory

**Строго типизованные объекты DataSet**

Строго типизованные объекты DataSet (как и подразумевает их название) позволяют взаимодействовать с внутренними таблицами объектов DataSet, используя для этого специальные свойства, методы и события базы данных, а не обобщенное свойство Tables. Выбрав View→Class View из меню в Visual Studio 2005, вы увидите, что мастер создал новый тип CarsDataSet, полученный из DataSet. Как видно из рис. 22.24, этот тип класса определяет ряд членов, позволяющих выбрать, изменить или обновить содержимое.

При выполнении мастером своей задачи он помещает в файл \*.Designer.cs член-переменную типа CarsDataSet (именно этот член используется для события Load формы).

partial class MainForm {

 …

 private CarsDataSet **CarsDataSet;**

}



Рис. 22.24. Строго типизованный объект DataSet

**Автоматически генерируемый компонент данных**

В дополнение к строго типизованному объекту DataSet, мастер генерирует компонент данных (в данном случае с именем InventoryTableAdapter), инкапсулирующий соответствующее соединение, адаптер данных и объекты команд, которые используются при взаимодействии с таблицей Inventory.

public partial class InventoryTableAdapter : System.ComponentMоdel.Component {

**// Поля данных для доступа к данным.**

 private System.Data.SqlClient.SqlDataAdapter m\_adapter;

 private System.Data.SqlClient.SqlConnection m\_connection;

 private System.Data.SqlClient.SqlCommand[] m\_commandCollection;

 …

}

Также этот компонент определяет пользовательские методы Fill() и Update(). настроенные на работу с вашим объектом CarsDataSet, в дополнение к множеству членов, используемых для вставки, обновления и удаления строк внутренней таблицы Inventory. Заинтересованные читатели могут самостоятельно выяснить детали реализации каждого из этих членов. При этом можно надеяться, что после всех усилий, которые вы затратили на освоение материала этой главы, соответствующий этим членам программный код не должен казаться вам совершенно незнакомым.

**Замечание**. Больше об объектной модели ADO.NET, а также о соответствующие мастерах Visual Studio 2005, вы узнаете из книги Сахила Малика, *Microsoft ADO.NET 2.0 для профессионалов*(пер. с англ, ИД "Вильямс", 2006 г.).

**Резюме**

ADO.NET является новой технологией доступа к данным, специально разработанной с учетом несвязных многоуровневых приложений. Пространство имен System.Data содержит большинство базовых типов, которые могут потребоваться для программного взаимодействия со строками, столбцами, таблицами и представлениями. При изучении материала главы вы могли убедиться в том, что в составе дистрибутива .NET предлагается множество поставщиков данных, позволяющих использовать как связный, так и несвязный уровни ADO.NET.

Используя объекты соединения, объекты команд и объекты чтения данных связного уровня, вы можете выбирать, обновлять, вставлять и удалять записи. Объекты команд поддерживают внутреннюю коллекцию параметров, которую можно использовать для повышения типовой безопасности SQL-запросов и которая оказывается очень полезной при запуске хранимых процедур.

Основным объектом несвязного уровня является DataSet. Этот тип является размещенным в памяти "представителем" любого числа таблиц и любого числа их необязательных взаимосвязей, ограничений и условий. "Красота" создания отношений на базе локальных таблиц заключается в том, что вы получаете возможность программно исследовать их, отключившись от удаленного хранилища данных.

Здесь же была рассмотрена роль адаптера данных. Используя соответствующий тип (и соответствующие свойства SelectCommand, InsertCommand, UpdateCommand и DeleteCommand), вы получаете возможность согласовывать изменения данных в DataSet с оригинальным хранилищем данных.

**ЧАСТЬ V. Web-приложения и Web-сервисы XML**

**ГЛАВА 23. Web-страницы и Web-элементы управления ASP.NET 2.0**

До сих пор все примеры приложений в этой книге касались консольных приложений и приложений Windows Forms. В этой главе и далее мы выясним, каким образом платформа .NET упрощает задачу построения приложений о интерфейсом на основе браузера, Но сначала мы обсудим ряд ключевых для Web понятий (таких, как HTTP, HTML, сценарий клиента и сервера), а также роль Web-сервера (включая сервер разработки ASP.NET, WebDev.Webserver.exe),

На основе полученной информации в оставшейся части главы мы сконцентрируемся на компонентах ASP.NET (включая усовершенствованную модель страницы с внешним кодом поддержки) и на использовании Web-элементов управления ASP.NET. Вы увидите, что ASP.NET 2.0 предлагает целый ряд новых элементов управления, новую модель "шаблона" страницы и новые возможности настройки Web-страниц.

**Роль HTTP**

Web-приложения очень сильно отличаются от традиционных приложений для настольных систем. Первым очевидным отличием является то, что любое реальное Web-приложение предполагает использование, как минимум, двух соединенных в сеть, машин (конечно, при разработке приложения вполне возможно, чтобы роли клиента и сервера играла одна машина). Задействованные машины должны согласовать использование определенного сетевого протокола для успешного осуществления отправки и приёма данных. Сетевым протоколом, соединяющим компьютеры в рассматриваемом нами случае, является протокол HTTP (Hypertext Transfer Protocol – протокол передачи гипертекста).

Когда машина-клиент запускает Web-браузер (такой, как Netscape Navigator, Mozilla Firefox или Microsoft Internet Explorer), генерируется HTTP-запрос доступа к конкретному ресурсу (например, к файлу \*.aspx или \*.htm) на удаленной машине-сервере. Протокол HTTP – это текстовый протокол, построенный на стандартной парадигме запросов и ответов. Например, при обращении к[http://www.IntertechTraining.com](http://www.intertechtraining.com/) программное обеспечение браузера использует Web-технологию, называемую сервисом DNS (Domain Name Service – служба имен доменов), которая позволяет превратить зарегистрированный адрес URL в 4-байтовое (32-разрядное) числовое значение (называемое IP-адресом). После этого браузер открывает сокет (обычно через порт с номером 80) и посылает HTTP-запрос странице, используемой Web-узлом http://www.IntertechTraining.com по умолчанию. Осуществляющий хостинг Web-сервер получает поступающий HTTP-запрос, и указанный в запросе ресурс может содержать программную логику, способную прочитать значения, введенные клиентом (например, в окне текстового блока), чтобы сформировать HTTP-ответ, Разработчик Web-программы может использовать любые технологии (CGI, ASP, ASP.NET, сервлеты Java и т.д.), чтобы динамически генерировать содержимое HTTP-ответа. Затем браузер клиента отображает HTML-код, полученный от Web-сервера. На рис. 23.1 показана общая схема цикла запросов-ответов HTTP.

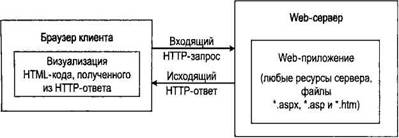


Рис. 23.1. Цикл запросов и ответов HTTP

Другой особенностью Web-разработки, заметно отличающей ее от программирования традиционных приложений, оказывается то, что протокол HTTP является сетевым протоколом, не сохраняющим состояние. Как только Web-сервер отправляет ответ клиенту, вся информация о предыдущем взаимодействии оказывается "забытой". Поэтому на вас, как Web-разработчика, возлагается задача принятия специальных мер, обеспечивающих "запоминание" соответствующей информации о клиентах, которые в настоящий момент оказываются зарегистрированными на вашем узле (такой информацией может быть, например, список товаров в корзине покупателя). В следующей главе вы сможете убедиться в том, что ASP.NET обеспечивает целый ряд способов обработки состояния, причем как стандартных для всех Web-платформ (это сеансовые переменные, файлы cookie и переменные приложения), так и новых (визуальные состояния, состояния элементов и кэш).

**Web-приложения и Web-серверы**

Под *Web-приложением*можно понимать коллекцию файлов (\*.htm, \*.asp, '\*.aspx, файлы изображений и т.д.) и связанных компонентов (например, таких как библиотека программного кода .NET), хранимых в отдельном семействе каталогов на данном Web-сервере. Как будет показано в главе 24, Web-приложения имеют специфический цикл существования и поддерживают множество специальных событий (например, события начальной загрузки и окончательного завершения работы), которые вы можете обработать.

*Web-сервер*- это программный продукт, обеспечивающий хостинг для ваших Web-приложений, и, как правило, предлагающий целый ряд сопутствующих сервисов, таких как, например, интегрированные службы безопасности, поддержка FTP (File Transfer Protocol – протокол передачи файлов), службы обмена почтовыми сообщениями и т.д. Web-сервером производственного уровня является сервер IIS (Internet Information Server – информационный сервер Интернет) от Microsoft, который, как вы можете догадаться, предлагает внутреннюю поддержку и "классических" Web-приложений ASP, и Web-приложений ASP.NET.

При создании Web-приложений ASP.NET вам потребуется взаимодействие с IIS. И здесь важно подчеркнуть, что сервер IIS по умолчанию при установке Windows Server 2003 или Windows XP Professional Edition *не устанавливается*(а среда Windows XP Home Edition поддержку IIS не предлагает вообще). Поэтому, в зависимости от конфигурации вашей машины разработки, вам, возможно, придется установить IIS вручную. Для этого откройте окно Установка и удаление программ (Add/Remove Program) из папки Панель управления (Control Panel) и выберите в нем Установка компонентов Windows (Add/Remove Windows Components).

**Замечание**. Сервер IIS лучше установить до установки .NET Framework. Если установить IIS после установки .NET Framework, то Web-приложения ASP.NET не будут выполняться корректно (вы увидите только пустые страницы). К счастью, можно настроить IIS на поддержку .NET-приложе-ний с помощью запуска утилиты командной строки aspnet\_regiis.exe (с флагом /i).

В предположении о том, что на вашей рабочей станции сервер IIS установлен должным образом, вы сможете взаимодействовать с IIS из папки Администрирование (размещенной в папке Панель управления). Для работы с материалом этой шалы нас будет интересовать только узел Web-узел по умолчанию (Default Web Site), рис. 23.1.



Рис. 23.2. Окно оснастки IIS

**Работа с виртуальными каталогами IIS**

Одна инсталляция IIS способна обслуживать множество Web-приложений, каждое из которых размещается в своем *виртуальном каталоге.*Каждый виртуальный каталог проецируется в физический каталог на локальном жестком диске. Так, если вы создадите новый виртуальный каталог с именем CarsRUs, внешние пользователи смогут просматривать этот узел, используя, например, адрес URL http://www.CarsRUs.com (в предположении о том, что IР-адрес вашего узла имеет всемирную регистрацию). В фоновом режиме виртуальный каталог отображается в соответствующий физический корневой каталог, например, в C:\inetpub\www-root\AspNetCarsSite, в котором находится содержимое Web-приложения.

При создании Web-приложения ASP.NET с помощью Visual Studio 2005 вы имеете возможность генерировать новый виртуальный каталог для текущего Web-узла. Вы также можете создать виртуальный каталог вручную. Для примера предположим, что нам нужно создать простое Web-приложение с именем Cars. Первым шагом при этом оказывается создание на машине новой папки (например, папки C:\CodeTests\CarsWebSite), которая будет содержать коллекцию файлов, компонующих новый узел.

Затем нужно создать новый виртуальный каталог для узла Cars. Просто щелкните в окне IIS правой кнопкой мыши в строке Веб-узел по умолчанию и выберите Создать→Виртуальный каталог из появившегося контекстного меню. Будет запущен интегрированный мастер создания виртуальных каталогов. Перейдите от окна приветствия к следующему окну и укажите для вашего Web-узла подходящий псевдоним (Cars). Далее вас попросят указать физическую папку на жестком диске, которая содержит файлы и изображения, используемые для этого узла (для нашего примера это папка C:\CodeTests\CarsWebSite).

На заключительном шаге мастер запрашивает информацию о правах доступа к новому виртуальному каталогу (разрешение доступа к файлам дли чтения/записи, обзора файлов с помощью Web-браузера, запуска выполняемых файлов и т.д.), Для нашего примера вполне подойдет вариант выбора, предлагаемый мастером по умолчанию (тем более, что вы в любое время можете изменить эти настройки, открыв окно свойств с помощью щелчка правой кнопкой мыши в любом из окон, интегрированных в IIS). По завершении работы мастера вы увидите в окне IIS новый виртуальный каталог (рис. 23.3).



Рис. 23.3. Виртуальный каталог Cars

**Сервер разработки ASP.NET 2.0**

Комплект поставки ASP.NET 2.0 содержит "облегченную" версию Web-сервера под названием WebDev.WebServer.exe. Эта утилита позволяет разработчику осуществлять хостинг Web-приложений ASP.NET 2.0 за границами IIS. С помощью этого инструмента вы можете строить и проверять Web-страницы из любого каталога на своей машине (что очень удобно при разработке сценариев в группе разработчиков и при создании Web-программ ASP.NET 2.0 в среде ОС Windows XP Home Edition, которая не поддерживает US).

**Замечание**. Сервер WebDev.WebServer.exe нельзя использовать для тестирования "классических" Web-приложений ASP.

При построении Web-узла с помощью Visual Studio 2005 вы имеете возможность использовать WebDev.WebServer.exe для обслуживания создаваемых страниц. Но вы также имеете возможность взаимодействовать с этим инструментом вручную из командной строки .NET. Если ввести команду

WebDev.WebServer.exe -?

вы увидите окно сообщения, в котором будут описаны действительные опции командной строки. В сущности, вам нужно указать неиспользуемый порт с помощью опции /port:, корневой каталог Web-приложения с помощью опции /path: и необязательный виртуальный путь с помощью опции /vpath: (если вы не укажете значение /vpath:, по умолчанию используется значение /). Рассмотрим следующий пример.

WebDev.WebServer.exe /port: 12345 /path:"C:\CodeTests\CarsWebSite"

После ввода этой команды вы можете запустить свой любимый Web-браузер для запроса соответствующих страниц. Так, если в папке CarsWebSite содержится файл с именем MyPage.aspx, вы можете ввести следующий адрес URL.

http://localhost:12345/CarsWebSite/MyPage.aspx

Во многих примерах из этой и следующей глав WebDev.WebServer.exe будет использоваться через Visual Studio 2005. Следует учитывать то, что этот Web-сервер не предназначен для хостинга Web-приложений производственного уровня, он предназначен исключительно для целей разработки и тестирования.

**Замечание**. Проект Mono (см. главу 1) предлагает бесплатное расширение ASP.NET для Web-сервера Apache. За более подробной информацией обратитесь по адресу: [http://www.mono-project.com/ASP.NET](http://www.mono-project.com/asp.net)

**Роль HTML**

Сконфигурировав каталог для своего Web-приложения, вы должны создать и его содержимое. Напомним, что *Web-приложение*- это просто термин используемый для обозначения множества файлов, обеспечивающих функционирование узла, Значительная часть этих файлов будет содержать синтаксические лексемы, определенные в рамках HTML (Hypertext Markup Language – язык гипертекстовой разметки). HTML – это стандартный язык, используемый для описания того, как в окне браузера клиента должна выполняться визуализация буквального текста, изображений, внешних ссылок и различных элементов графическом) интерфейса.

Этот специальный аспект Web-разработки является одной из главных причин столь распространенной нелюбви программистов, которую они испытывают к разработке Web-программ. И хотя современные средства Web-разработки (включан Visual Studio 2005) и платформы (такие как ASP.NET) генерируют большинство HTML-кода автоматически, сегодня для успешной работы с ASP.NET все еще важно хорошо понимать этот язык. Данный раздел, конечно же, ни в коей мере не претендует на охват всех аспектов HTML, но давайте рассмотрим основные.

**Структура HTML-документа**

Файл HTML состоит из множества дескрипторов, описывающих представление данной Web-страницы. Как и следует ожидать, базовая структура любого HTML-документа примерно одинакова. Например, файлы \*.htm (или, альтернативно, файлы \*.html) открываются и закрываются дескрипторами ‹html› и ‹/html›, обычно в них определяется раздел ‹body› и т.д. Следует иметь в виду, что HTML не чувствителен к регистру символов. Поэтому для браузера ‹HTML›, ‹html› и ‹Html› оказываются идентичными.

Для демонстрации некоторых базовых возможностей HTML откройте Visual Studio 2005, создайте пустой HTML-файл, выбрав File→New→File из меню, и сохраните этот файл под именем default.htm в каталоге C:\CodeTests\CarsWebSite. Наша исходная разметка весьма незамысловата.

‹html›

 ‹body›

 ‹/body›

‹/html›

Дескрипторы ‹html› и ‹/html› используются для обозначения начала и конца документа. Как вы можете догадаться, Web-браузер использует эти дескрипторы, чтобы выяснить, с какого места следует начать и где следует закончить обработку признаков форматирования, указанных в главной части документа. Почти все содержимое документа определяется в рамках дескриптора ‹body›. Чтобы немного "оживить" страницу, определите ее заголовок так, как показано ниже.

‹html›

**‹head›**

**‹title›Web-страница Cars‹/title›**

**‹/head›**

 ‹body›

 ‹/body›

‹/html›

Вы, наверное, догадались, что дескрипторы ‹title› используются для обозначения текстовой строки, которая при вызове этой страницы должна размещаться в строке заголовка окна Web-браузера.

**Разработка HTML-формы**

Реальное действие в файле \*.htm происходит в рамках элементов ‹form›. *HTML-форма*- это просто именованная группа связанных элементов пользовательского интерфейса, используемых для сбора данных пользовательского ввода, которые затем передаются Web-приложению по протоколу HTTP. He следует путать HTML-форму со всей областью окна браузера. Фактически HTML-форма представляет собой *логическое объединение*элементов, размещенных между дескрипторами ‹form› и ‹/form›.

‹html›

 ‹head›

  ‹titlе›Web-страница Cars‹/title›

 ‹/head›

 ‹body›

  ‹form id="defaultPage" name="defaultPage"›

**‹!-- Место для Web-содержимого -**-›

  ‹/form›

 ‹/body›

‹/html›

Для id и name этой формы указано значение default Page. Как правило, открывающий дескриптор ‹form› задает также атрибут aсtion, указывающий адрес URL, по которому следует передать данные формы, и метод передали этих данных (POST или GET). Эти возможности дескриптора ‹form› мы рассмотрим чуть позже. Пока чего давайте выясним, какие элементы могут размещаться в HTML-форме. В панели инструментов Visual Studio 2005 предлагается специальный раздел HTML, в котором сгруппированы связанные с HTML элементы управления (рис. 23.4).



Рис. 23.4. Раздел HTML в окне панели инструментов

**Создание пользовательского интерфейса на базе HTML**

Перед добавлением HTML-элементов в HTML-форму важно заметить, что Visual Studio 2005 позволяет редактировать содержимое файлов \*.htm с помощью интегрированного HTML-редактора и окна свойств. При выборе DOCUMENT в окне свойств (рис. 23.5) вы получаете возможность настройки ряда параметров HTML-страницы, например цвета ее фона.

Теперь измените раздел *‹*body*›*файла default.htm так, чтобы отобразить пользователю приглашение ввести имя и пароль, и установите для фона тот цвет, который вам нравится (вы можете вводить и форматировать текстовое содержимое непосредственно в окне НТМL-редактора).

‹html›

 ‹head›

  ‹titlе›Web-страница Cars‹/title›

 ‹/head›

 ‹body bgcolor="NavajoWhite"›

  ‹!-- **Приглашение ввода для пользователя -**-›

  ‹h1 align="center"›Страница входа в систему Cars‹/h1›

  ‹р align="center"›‹br›Введите ‹i›имя пользователя‹/i› и ‹i›пароль‹/i›.‹/р›

  ‹form id="defaultPage" name="defaultPage"› ‹/form›

 ‹/body›

‹/html›

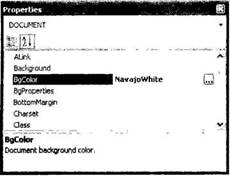


Рис. 23.5. Редактирование HTML-документа средствами VS .NET

Теперь давайте построим саму HTML-форму. Вообще говоря, каждый HTML-элемент описывается с помощью атрибута name (соответствующее значение используется для программной идентификации элемента) и атрибута type (это значение задает вид элемента интерфейса, который вы хотите поместить в рамки декларации ‹form›). В зависимости оттого, с каким элементом интерфейса вы работаете, в окне свойств появляется дополнительные атрибуты, присущие данному конкретному элементу.

Пользовательский интерфейс, который мы собираемся здесь построить, будет содержать два текстовых поля ввода (одно из которых – типа Password) и два кнопочных типа (один для подачи запроса с данными формы, а другой – для сброса данных формы в значения по умолчанию).

‹!-- Построение формы для получения информации от пользователя --›

‹form name="defaultPage" id="defaultPage"›

 ‹P align="center"›Имя пользователя:

  ‹input id="txtUserName" **type="text"**NAME="txtUserName"›

 ‹/P›

 ‹P align="center"›пароль:

  ‹input name="txtPassword" **type="password"**ID="txtPassword"›

 ‹/P›

 ‹P align="center"›

  ‹input name="btnSubmit" **type="submit"**value="Отправить" io="btnSubmit"›

  ‹input name="btnReset" **type="reset"**value="C6poc" ID="btnReset"›

‹/form›

Обратите внимание на то, что здесь для каждого элемента назначены соответствующие значения name и id (txtUserName, txtPassword, btnSubmit и btnReset). Еще более важно то, что каждый элемент ввода имеет дополнительный атрибут type, который ясно идентифицирует их как вполне определенные элементы пользовательского интерфейса. Например, type="reset" указывает на автоматическую очистку всех полей формы и присвоение им начальных значений, type="password" – маскированный ввод пароля, a type="submit" – отправку данных формы получателю. На рис. 23.6 показан вид получившейся страницы.



Рис. 23.6. Исходный вид страницы, сохранённой в файле default.htm

**Роль сценариев клиента**

Данный файл \*.htm может содержать блок кода сценария, который будет помещен в ответный поток и обработан браузером, запросившим этот поток. Есть две главные причины, по которым используются сценарии клиента.

• Проверка пользовательского ввода перед отправкой данных Web-серверу.

• Взаимодействие с моделью DOМ целевого браузера.

В отношении первого пункта следует понимать, что "наследственной" проблемой Web-приложений является необходимость частых обращений к серверу (называемых вторичными обращениями) для обновления HTML-кода, отображаемого в окне браузера. И хотя вторичных обращений полностью избежать нельзя, всегда нужно стремиться минимизировать сетевой обмен. Одним из подходов, уменьшающих количество циклов сетевого обмена, является использование сценария клиента для проверка правильности пользовательского ввода перед отправкой данных формы Web-серверу. Если обнаруживается ошибка (например, не указаны данные в одном из обязательных полей), можно предложить пользователю исправить ошибку, не посылая данные Web-серверу напрасно. (В конце концов, ничто не раздражает пользователя больше, чем отправка данных по медленной связи только для того, чтобы получить в ответ совет исправить ошибки ввода!)

В дополнение к проверке пользовательского ввода, сценарии клиента могут также использоваться для взаимодействия с объектной моделью DOM (Document Object Model – объектная модель документов) браузера. Большинство коммерческих браузеров предлагает множество объектов, которые можно использовать для управления поведением браузера. Главным раздражающим фактором здесь является то, что различные браузеры предлагают подобные, но не идентичные объектные модели. Поэтому запущенный вами сценарий клиента, взаимодействующий с DOM, может работать по-разному в разных браузерах.

**Замечание.**ASP.NET поддерживает свойство HttpRequest.Browser, которое позволяет в среде выполнения определить возможности браузера, отправившего текущий запрос.

Имеется множество языков сценариев, которые могут использоваться для написания программного кода сценариев клиента. Двумя наиболее популярными из них являются VBScript и JavaScript. Язык VBScript представляет собой подмножество языка программирования Visual Basic 6.0. Следует подчеркнуть, что Microsoft Internet Explorer (IE) – это единственный Web-браузер, имеющий встроенную поддержку VBScript клиента. Поэтому если вы хотите, чтобы ваши HTML-страницы работали корректно в *любом*коммерческом Web-браузере, для программной логики сценариев клиента лучше VBScript не использовать.

Другим популярным языком сценариев является JavaScript. Здесь следует подчеркнуть, что JavaScript никоим образом не является подмножество языка Java. Хотя JavaScript и Java имеют несколько схожий синтаксис, JavaScript нельзя безоговорочно отнести к семейству языков ООП, поэтому этот язык оказывается далеко не таким мощным, как Java. Но здесь важно то, что все современные Web-браузеры поддерживают JavaScript, что делает этот язык естественным кандидатом на роль языка сценариев клиента.

**Замечание.**Чтобы еще больше усложнить ситуацию, напомним также о JScript.NET – управляемом языке программирования, с помощью которого, используя подобный сценариям синтаксис, можно строить компоновочные блоки .NET.

**Пример сценария клиента**

Чтобы продемонстрировать роль сценариев клиента, давайте выясним, как можно выполнить перехват событий, посылаемых HTML-элементами пользовательского интерфейса клиента. Предположим, что вы добавили в свою HTML-страницу default.htm тип Button (с именем btnHelp), которая должна предоставить пользователю возможность увидеть информацию справки. Чтобы выполнить перехват события Click для этой кнопки, активизируйте окно просмотра HTML и выберите имя кнопки из левого раскрывающегося списка. Затем в правом раскрывающемся списке выберите событие onclick. В результате этого в определение атрибута нового типа Button будет добавлен атрибут onclick.

‹input id="btnHelp" type="button" Value="Help" language="javascript" **onclick="return btnHelp\_onclick()**"/›

Visual Studio 2005 также создаст пустую функцию JavaScript, которая будет вызываться при щелчке пользователя на кнопке. Чтобы отобразить окно сообщения клиента, нужно в пределах этой заглушки просто использовать метод alert().

‹script language="javascript" type="text/javascript"›

‹!--

function **btnHelp\_onclick()**{

 alert("Это не так уж трудно. Просто щелкните на кнопке Отправить!");

}

-->

‹/script›

Обратите внимание на то, что блок сценария помещен в рамки HTML-комментария (‹!-- --›). Причина этого очень проста. Если ваша страница окажется в браузере, не поддерживающем JavaScript, программный код будет интерпретирован как комментарий и потому проигнорирован. Конечно, возможности вашей страницы будут более узкими, но зато ваша страница не будет полностью отвергнута браузером.

**Контроль допустимости вводимых данных**

Теперь давайте добавим в нашу страницу default.htm клиентскую поддержку контроля вводимых в форму данных. Нам нужно, чтобы при щелчке пользователя на кнопке Отправить вызвалась функция JavaScript, которая проверяла бы каждый текстовый блок на пустые значения. При наличии пустого значения должно появиться окно сообщения с указанием ввести правильные данные. Сначала обработайте событие onclick для кнопки Отправить.

‹input name="btnSubmit" type="Submit" value="Submit" id="btnSubmit" languaege="javascript" onclick*=*"return btnSubmit\_onclick()"›

Этот обработчик события реализуйте так, как показано ниже.

function btnSubmit\_onclick() {

**// Если пользователь о чем-то забыл, отобразить сообщение.**

 if ((defaultPage.txtUserName.value == "") || (defaultPage.txtPassword.value == "")) {

  alert("Следует указать имя пользователя и пароль!");

  return false;

 }

 return true;

}

Теперь вы можете открыть свой любимый браузер, перейти к странице default.htm в виртуальном каталоге Cars и проверить работу вашего сценария клиента.

http://localhost/Cars/default.htm

**Подача запроса формы (GET и POST)**

Теперь, когда у вас есть простая HTML-страница, мы должны выяснить, как передать данные формы обратно Web-серверу для обработки. При построении HTML-формы в открывающем дескрипторе ‹form› обычно задается атрибут action, указывающий получателя вводимых в форму данных. Возможными получателями могут быть почтовые серверы, другие файлы HTML, файлы ASP (как "классические", так и .NET) и т.д. Для нашего примера мы используем "классический" файл ASP с именем ClassicAspPage.asp. Обновите свой файл default.htm, указав в нем следующие атрибуты в открывающем дескрипторе ‹form›.

‹form name="defaultPage" id="defaultPage" **action="http://localhost/Cars/ClassicAspPage.asp" method**= **"GET"›**

‹/form›

Добавленные атрибуты гарантируют, что при щелчке на кнопке Отправить данные формы будут отправлены файлу ClassicAspPage.asp с указанным URL. Указание method = "GET" для режима передачи означает, что данные формы присоединяются к строке запроса в виде набора пар имен и значений, разделенных символами амперсанда.

http://localhost/Cars/ClassicAspPage**.asp?txtUserName=Andrew&txtPassword=abcd123$**&btnSubmit=Submit

Другой метод передачи данных формы Web-серверу указывается с помощью method = "POST".

‹form name="defaultPage" id="defaultPage" action="http://localhost/Cars/ClassicAspPage.asp" **method = "POST"›**

‹/form›

В этом случае данные формы не присоединены к строке запроса, а записываются в отдельной строке в рамках HTTP-заголовка. При использовании POST данные формы будут невидимы для внешнего наблюдателя. Еще более важно то, что POST не имеет ограничений на длину символьных данных (многие браузеры выдвигают ограничения на длину запросов с использованием GET). Пока что для отправки данных формы странице-получателю \*.asp мы будем использовать HTTP-метод GET.

**Создание "классической" ASP-страницы**

"Классическая" ASP-страница является комбинацией HTML и программного кода сценария сервера. Если вы никогда не работали с ASP, вам будет полезно знать, что целью использования ASP является динамическое построение HTML-кода с помощью *сценария сервера*и небольшого набора классических COM-объектов. Например, вы можете иметь серверный блок VBScript (или JavaScript), который читает таблицу из некоторого источника данных, используя классическую технологию ADO, и возвращает строки в виде HTML-таблицы общего вида.

В нашем примере ASP-страница использует внутренний COM-объект Request, чтобы прочитать введенные в форму данные (присоединенные к строке запроса) и возвратить их обратно вызывающей стороне в виде эхо (не слишком впечатляет, но поставленная задача будет выполнена). Для сценария сервера мы используем VBScript (что обозначено директивой language).

С этой целью создайте новый HTML-файл и сохраните его с именем ClassicAspPage.asp в той папке, куда проецируется ваш виртуальный каталог (например, в папке C:\CodeTests\CarsWebSite). Реализуйте эту страницу так, как предлагается ниже.

**‹%@ language="VBScript" %›**

‹html›

 ‹head›

  ‹titlе›Страница Cars‹/title›

 </head>

 ‹body›

  ‹h1 align="center"›Вот что вы нам прислали:‹/h1›

  ‹P align="center"›‹b›Имя пользователя: ‹/b›

  ‹%= **Request.QueryString**("txtUserName") %›‹br›

  ‹b›Пароль: ‹/b›

  ‹%= **Request.QueryString**("txtPassword") %›‹br›

 ‹/body›

</html>

Здесь COM-объекта Request ASP используется для вызова метода QueryString() с целью анализа значений, содержащихся в HTML-элементах и переданных с помощью method = "GET". Обозначение ‹%=… %› является сокращением для требования "вставить указанное непосредственно в исходящий HTTP-ответ". Чтобы достичь большей гибкости, вы могли бы взаимодействовать с COM-объектом Response в рамках всего блока сценария (обозначаемого знаками ‹% %›). В этом здесь необходимости нет, однако вот простой пример.

**‹%**

 Dim pwd

 pwd = **Request.QueryString**("txtPassword")

**Response.Write**(pwd)

**%›**

Очевидно, что объекты Request и Response классической схемы ASP предлагают целый ряд дополнительных членов, кроме показанных ниже. К тому же, в рамках классического подхода ASP определяется небольшой набор дополнительных COM-объектов (Session, Server, Application и т.д.), которые вы тоже можете использовать при построении Web-приложения.

**Замечание**. В ASP.NET эти COM-объекты официально не существуют. Однако вы увидите, что базовый класс System.Web.UI.Page определяет свойства с идентичными именами, возвращающие объекты с аналогичными возможностями.

Чтобы проверить программную логику ASP в нашем случае, просто загрузите страницу default.htm в браузер и введите в форму данные. После обработки соответствующего сценария на Web-сервере вы получите новый (динамически сгенерированный) HTML-код (рис. 23.7).

**Ответ на отправку POST**

В настоящий момент для отправки данных формы целевому файлу \*.asp в вашем файле default.htm указан HTTP-метод GET. При использовании этого подхода значения, содержащиеся в элементах управления графического интерфейса, присоединяются в конец строки запроса. Здесь важно подчеркнуть, что ASP-метод Request.QueryString() способен извлекать данные, переданные *только*с помощью метода GET.

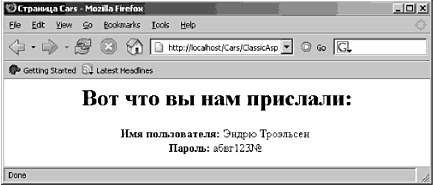


Рис. 23.7. Динамически сгенерированная HTML-страница

Чтобы представить данные формы Web-pecypcy, используя HTTP-метод POST, можно использовать коллекцию Request.Form, чтобы прочитать значения на сервере, например:

‹body›

 ‹h1 align="center"›Bот что вы нам прислали:‹/h1›

 ‹Р align="center"›

 ‹b›Имя пользователя: ‹/b›

 ‹%= **Request.Form**("txtUserName") %› ‹br›

 ‹b›Пароль: ‹/b›

 ‹%= **Request.Form**("txtPassword") %› ‹br›

‹/body›

На этом наше обсуждение основ Web-разработки завершено. Надеюсь, что даже если вы до сих пор не имели никакого опыта разработки Web-приложений, теперь вы понимаете основные принципы создания таких приложений. Перед выяснением того, как платформа .NET совершенствует существующие на сегодня подходы, давайте потратим немного времени, чтобы "покритиковать" классический подход ASP.

**Исходный код**. Файл примера ClassicAspPage размещен в подкаталоге, соответствующем главе 23.

**Проблемы классической технологии ASP**

С помощью классической технологии ASP (Active Server Pages – активные серверные страницы) создано очень много популярных Web-узлов, но эта архитектура имеет свои ограничения. Возможно, самым большим ограничением классической технологии ASP как раз и является то, что делает эту технологию такой мощной – это языки сценариев сервера. Языки сценариев, такие как VBScript и JavaScript, являются интерпретируемыми, не предусматривающими определения типов данных и не способствующими созданию надежных объектно-ориентированных программных конструкций.

Второй проблемой классической технологии ASP оказывается то, что программный код страницы \*.asp не является строго модульным. Поскольку ASP представляет собой комбинацию HTML и сценариев в рамках *одной*страницы, большинство Web-приложений ASP оказывается "странной смесью" двух совсем разных подходов в программировании. И хотя классическая технология ASP позволяет разделить Многократна используемый программный код на отдельные включаемые в проект файлы, лежащая в основе такого разделения объектная модель не обеспечивает истинное разграничение обязанностей. В идеале каркас Web-разработки должен позволить программной логике представления (т.е. дескрипторам HTML) существовать независимо от программной логики реализации (т.е. программного кода, реализующего функциональные возможности приложения).

Наконец, еще одной проблемой является то, что классическая технология ASP требует использования большого количества шаблонов и весьма избыточного программного кода сценариев, который может повторяться от проекта к проекту. Почти всем Web-приложениям необходимо осуществлять контроль пользовательского ввода, обновлять состояние HTML-элементов перед отправкой HTTP-ответа, генерировать HTML-таблицы данных и т.д.

**Главные преимущества ASP.NET 1.х**

Уже первая главная реализация ASP.NET (версии 1.x) предложила фантастические возможности преодоления ограничений, присущих классической технологии ASP. По сути, платформа .NET дала начало использованию следующих подходов.

• ASP.NET 1.x предлагает модель, основанную на использовании *внешнего кода поддержки*и позволяющую отделить логику представления от логики реализации.

• Страницы ASP.NET 1.x представляют собой скомпилированные компоновочные блоки .NET, а не интерпретируемые строки языка сценариев, которые обрабатываются значительно медленнее.

• Web-элементы управления позволяют программисту строить Web-приложения с графическим интерфейсом приблизительно так же, как и приложения Windows Forms.

• Web-элементы ASP.NET автоматически обновляют своё состояние при вторичных запросах, для чего используется скрытое поле формы, имеющее имя \_\_VIEWSTATE.

• Web-приложения ASP.NET являются полностью объектно-ориентированными и используют CTS (Common Type System – общая система типов).

• Web-приложения ASP.NET легко конфигурировать с помощью стандартных средств IIS или с помощью файла конфигурации Web-приложения (Web.config).

Технология ASP-NET 1.x была большим шагом в правильном направлении, но ASP.NET 2.0 обеспечивает дополнительные преимущества.

**Главные преимущества ASP.NET 2.0**

ASP.NET 2.0 предлагает ряд новых пространств имен, типов, утилит и подходов в разработке Web-приложений .NET. Вот их неполный список.

• В ASP.NET 2.0 для разрабатываемого и тестируемого Web-узла уже не требуется хостинг в IIS. Теперь вы можете разместить свой узел в любом каталоге на жестком диске.

• ASP.NET 2.0 поставляется с большим набором новых Web-элементов (элементы управления безопасностью, элементы управления данными, элементы пользовательского интерфейса и т.д.), дополняющим набор элементов управления ASP.NET 1.x.

• В ASP.NET 2.0 поддерживаются *шаблоны страниц,*которые позволяют создать общий шаблон интерфейса для множества связанных страниц.

• В ASP.NET 2.0 поддерживаются *темы,*которые предлагают декларативный метод изменения внешнего вида всего Web-приложения.

• В ASP.NET 2.0 поддерживаются *Web-части,*которые могут использоваться для того, чтобы конечный пользователь мог настроить внешний вид Web-страницы.

• В ASP.NET 2.0 поддерживается Web-утилита конфигурации и управления, которая осуществляет управление файлами Web.config.

Если бы здесь ставилась задача описать все новые возможности ASP.NET 2.0, эта книга была бы в два раза больше. Поскольку тема Web-разработки не является единственной темой рассмотрения, для получения более конкретной информации о том, что здесь не представлено, обратитесь к документации .NET Framework 2.0.

**Пространства имен ASP.NET 2.0**

В библиотеках базовых классов .NET 2.0 предлагается не менее 34 пространств имен, имеющих отношение к Web. Всю эту совокупность пространств имен можно разбить на четыре главные группы.

• Базовые функциональные возможности (типы, обеспечивающие взаимодействие с HTTP-запросами и HTTP-ответами, инфраструктура Web-форм, поддержка тем и профилирования, Web-части и т.д.)

• Web-формы и HTML-элементы

• Web-разработка для мобильных платформ

• Web-сервисы XML

В этой книге тема разработки .NET-приложений (ни Web-приложений, ни каких-то других) для мобильных систем не рассматривается, но роль Web-сервисов XML будет обсуждаться в главе 25. В табл. 23.1 предлагаются описания некоторых базовых пространств имен ASP.NET 2.0.

Таблица 23.1. Пространства имен ASP.NET для Web

|  |  |
| --- | --- |
| **Пространства имен** | **Описание** |
| System.Web | Определяет типы, обеспечивающие коммуникацию браузера и Web-сервера (в частности, возможности запроса и ответа, обработки файлов cookie и передачи файлов) |
| System.Web.Caching | Определяет типы, обеспечивающие возможность кэширования для Web-приложения |
| System.Web.Hosting | Определяет типы, позволяющие строить пользовательские хосты для среды выполнения ASP.NET |
| System.Web.Management | Определяет типы, обеспечивающие управление и контроль правильности функционирования Web-приложения ASP.NET |
| System.Web.Profile | Определяет типы, используемые для работы с пользовательскими профилями ASP.NET |
| System.Web.Security | Определяет типы, позволяющие программно обеспечить безопасность узла |
| System.Web.SessionState | Определяет типы, обеспечивающие поддержку информации состояния для каждого пользователя (например, на основе использования сеансовых переменных состояния) |
| System.Web.UI Sуstem.Web.UI.WebControls System.Web.UI.HtmlControls | Определяют ряд типов, позволяющих создавать для Web-приложений программы клиента с графическим пользовательским интерфейсом |

**Модель программного кода Web-страницы ASP.NET**

Web-страницы ASP.NET могут строиться с использованием одного из двух подходов. Вы можете создать один файл *\**.aspx, который будет содержать комбинацию программного кода сервера и HTML (по примеру классической технологии ASP). Для создания такого файла используется модель одномодульной страницы, когда программный код сервера размещается в контексте ‹script›, но сам этот программный код непосредственно *не является*программным кодом сценария (например, на языке VBScript или JavaScript). Вместо этого операторы программного кода в рамках блока ‹script› записываются на любом из управляемых языков (C#, Visual Basic .NET и т.д.).

Если создаваемая страница содержит очень мало программного кода (и очень много HTML-кода), модель одномодульной страницы окажется лучшим вариантом выбора, так как перед вами а одном унифицированном файле \*. aspx будут и программный код, и разметка. Размещение программного и HTML-кода в одном файле \*.aspx обеспечивает и другие преимущества.

• Страницы, созданные в рамках одномодульной модели, проще инсталлировать и предоставлять другим разработчикам.

• Ввиду отсутствия зависимости между файлами, одномодульную страницу проще переименовать.

• Поддержка и обслуживание файлов исходного кода оказываются более простыми, поскольку все действие происходит в одном файле.

Подход, принятый в Visual Studio 2005 по умолчанию (при создании нового проекта Web-узла), использует так называемую технологию *внешнего кода поддержки*(code-behind), предполагающую отделение программного кода от HTML-логики представления и размещение их в двух разных файлах. Эта модель исключительно хорошо работает в тех случаях, когда ваши страницы содержат большие объемы программного кода или в процессе разработки Web-узла принимают участие много разработчиков. Модель, основанная на использовании внешнего кода поддержки, имеет несколько преимуществ.

• Ввиду совершенного разделения HTML-разметки и программного кода, становится возможным, чтобы созданием разметки занимались дизайнеры, а созданием программного кода C# – программисты.

• Программный код не предъявляется дизайнерам страницы и другим разработчикам, занимающимся только разметкой страницы (вы, наверное, догадываетесь, что разработчики HTML обычно не проявляют большого интереса к огромным объемам программного кода C#).

• Файлы программного кода могут использоваться в множестве файлов \*.aspx.

Выбранный вами подход *не*влияет на производительность полученного результата. Также следует подчеркнуть, что одномодульная модель \*.aspx, которая в .NET 1.x провозглашалась *неприемлемой,*теперь такой не считается. Теперь во многих Web-приложениях ASP.NET 2.0 при построении узлов используются преимущества обеих указанных выше моделей.

**Модель одномодульной страницы**

Сначала мы рассмотрим модель одномодульной страницы. Нашей целью является построение файла \*.aspx (с именем Default.aspx), который будет отображать таблицу Inventory базы данных Cars (созданной в главе 22). Вполне возможно создать эту страницу только с помощью редактора Блокнот, но Visual Studio 2005 может упростить процесс построения с помощью средств IntelliSense, автоматического завершения программного кода и визуальных редакторов страницы.

Сначала откройте Visual Studio 2005 и создайте новую Web-форму, выбрав File→New→File из меню (рис. 23.8).

После загрузки страницы в среду разработки обратите внимание на то, что в нижней части окна проектирования страницы есть две кнопки, позволяющие увидеть содержимое файла \*.aspx в двух разных вариантах. Выбрав кнопку Design, вы увидите окно визуального проектирования, в котором вы можете строить пользовательский интерфейс страницы во многом подобно тому, как вы строили интерфейс формы приложения Windows Form (перетаскивая элементы управления на поверхность формы, изменяя настройки в окне свойств и т.д.). Если выбрать кнопку Source, вы увидите HTML-код и блоки ‹script›, из которых скомпонован данный файл \*.aspx.

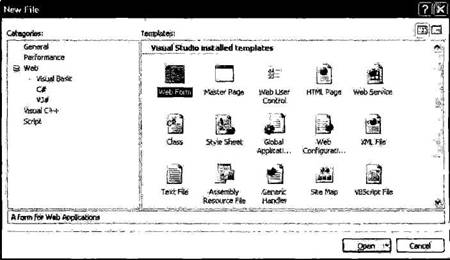


Рис. 23.8. Создание нового файла \*.aspx

**Замечание**. В отличие от более ранних версий Visual Studio, вид Source в Visual Studio 2005 предлагает полноценную поддержку IntelliSense и позволяет перетаскивание элементов пользовательского интерфейса непосредственно в окно HTML-кода.

В панели инструментов (окно Toolbox) Visual Studio 2005 откройте раздел Standard и перетащите элементы управления Button, Label и GridView в окно проектирования страницы (элемент GridView можно найти в разделе Data окна Toolbox). He пренебрегайте использованием окна свойств (или IntelliSense для HTML) при установке различных свойств элементов интерфейса и укажите для каждого Web-элемента подходящее имя с помощью свойства ID. На рис. 23.9 показан один из возможных вариантов оформления проекта (сдержанность здесь проявляется преднамеренно, чтобы минимизировать объем генерируемого кода разметки).

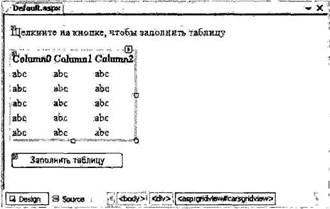


Рис. 23.9. Пользовательский интерфейс Default.aspx

Теперь щелкните на кнопке Source внизу окна и найдите в программном коде раздел ‹form› своей страницы. Обратите внимание на то, что Web-элементы управления определены с помощью дескриптора ‹asp:›, там же вы обнаружите набор пар имен и значений в соответствии с установками, сделанными вами в окне свойств.

‹form id="form1" runat="server"›

 ‹div›

**‹asp:Label**ID="lblInfo" runat="server" Техt="Щелкните на кнопке, чтобы заполнить таблицу"›

**‹/asp:Label›**

**‹asp:GridView**ID="carsGridView" runat="server"›

**‹/asp**:**GridView›**

**‹asp:Button**ID="btnFillData" runat="server" Text="Заполнить таблицу" /›

 ‹/div›

‹/form›

Подробности использования Web-элементов управления ASP.NET будут обсуждаться в главе позже. Пока что достаточно понять, что Web-элементы управления – это классы, обрабатываемые на Web-сервере и автоматически помещающие свое HTML-представление в исходящий HTTP-ответ (да, вам не нужно создавать соответствующий HTML-код!).

Кроме этого основного преимущества, Web-элементы ASP.NET поддерживают модель программирования, аналогичную Windows Forms, когда имена свойств, методов и событий имитируют их эквиваленты в Windows Forms. Для примера обработайте событие Click для типа Button либо с помощью окна свойств (используйте кнопку с пиктограммой молнии), находясь Visual Studio в режиме проектирования Web-формы, либо с помощью раскрывающихся списков, размещенных в верхней части окна просмотра программного кода (кнопка Source). В результате в определение Button будет добавлен атрибут OnClick, которому назначено имя обработчика события Click.

‹asp:Button ID="btnFillData" runat="server" Text="Заполнить таблицу" OnClick="btnFillData\_Click" /›

Кроме того, в ваш блок ‹script› добавляется обработчик события Click сервера (здесь обратите внимание на то, что входные параметры в точности соответствуют требованиям целевого делегата System.EventHandler),

‹script runat="server"›

protected void **btnFillData\_Click**(object sender, EventArgs e) {}

‹/script›

Реализуйте серверную часть обработчика событий так, чтобы использовался объект чтения данных ADO.NET для заполнения GridView. Также добавьте директиву импорта (подробнее об этом чуть позже), которая укажет, что вы используете пространство имен System.Data.SqlClient. Вот остальная часть соответствующей программной логики страницы файла Default.aspx.

‹%@Page Language="C#" %›

**‹%@Import Namespace = "System.Data.SqlClient" %›**

…

‹script runat="server"›

protected void btnFillData\_Click(object sender, EventArgs е) {

 SqlConnection: sqlConn = new SqlConnection("Data Source=.;Initial Catalog=Cars;UID=sa;PWD=");

 sqlConn.Open();

 SqlCommand cmd = new SqlCommand("Select \* From Inventory", sqlConn);

 carGridView.DataSource = cmd.ExecuteReader();

 carsGridView.DataBind();

 sqlConn.Close();

}

*‹/*script*›*

‹html xmlns = "http://www.w3.org/1999/xhtml"›

 …

‹/html›

Перед тем как погрузиться в детали обсуждения формата этого файла \*.aspx, давайте выполним тестовый запуск страницы. Откройте окно командной строки Visual Studio 2005 и запустите утилиту WebDev.WebServer.exe, указав путь к сохраненному вами файлу Default.aspx.

webdev.webserver.exe /port:12345 /path:"C:\CodeTests\SinglePageModel"

Затем, открыв любой браузер, введите следующий адрес URL.

http://localhost:12345/

При загрузке страницы вы сначала увидите только типы Label и Button. Но когда вы щелкнете на кнопке, Web-серверу будет направлен вторичный запрос, в результате которого Web-элементы управления получат обратно соответствующие HTML-дескрипторы. На рис. 23.10 показан результат визуализации нашей страницы в окне Mozilla Firefox.

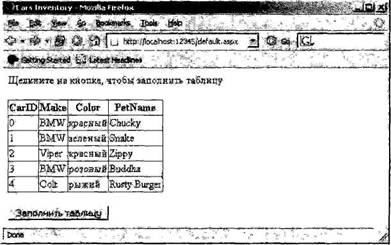


Рис. 23.10. Web-доступ к данным

Совсем просто, не правда ли? Но, как говорится, все зависит от мелочей, так что давайте рассмотрим немного подробнее композицию файла \*.aspx.

**Директива ‹%@Page%›**

Прежде всего следует отметить то, что файл \*.aspx обычно открывается набором директив. Директивы ASP.NET всегда обозначаются маркерами ‹%@ XXX %› и могут сопровождаться различными атрибутами, информирующими среду выполнения ASP.NET о том, как обрабатывать соответствующие данные.

Каждый файл \*.aspx должен иметь, как минимум, директиву ‹%@Page%›, которая используется для определения управляемого языка, применяемого в рамках страницы (для этого используется атрибут language). Также директива ‹%@Page%› может определять имя соответствующего файла с внешним кодом поддержки (если таковой имеется), разрешать трассировку и т.д. Наиболее интересные атрибуты ‹%@Page%› описаны в табл. 23.2.

**Таблица 23.2.**Подборка атрибутов директивы ‹%@Page%›

|  |  |
| --- | --- |
| **Атрибут** | **Описание** |
| CompilerOptions | Позволяет определить любые флаги командной строки (представленные одной строкой), передаваемые компилятору при обработке страницы |
| CodePage | Указывает имя соответствующего файла с внешним кодом поддержки |
| EnableTheming | Индикатор поддержки тем ASP.NET 2.0 элементами управления данной страницы \*.aspx |
| EnableViewState | Индикатор поддержки состояния представления между запросами страницы (более подробно об этом говорится в главе 24) |
| Inherits | Определяет класс страницы, из которой получается данный файл \*.aspx; может быть любым классом, полученным из System.Web.UI.Page |
| MasterPageFile | Указывает шаблон страницы, используемый в паре с текущей страницей \*.aspx |
| Trace | Индикатор разрешения трассировки |

**Директива ‹%@Import%›**

В дополнение к директиве ‹%@Page%› файл \*.aspx может использовать различные директивы ‹%@Import%›, чтобы явно указать пространства имен, необходимые для текущей страницы. В нашем примере указано использование типов из пространства имен System.Data.SqlClient. Ясно, что при необходимости использования дополнительных пространств имен .NET нужно просто указать несколько директив ‹%@Import%›.

**Замечание.**Директива ‹%@lmport%› не является необходимой, если применяется модель страницы с внешним кодом поддержки. При использовании файла с внешним кодом поддержки для указания внешних пространств имен применяется ключевое слово using C#.

Опираясь на имеющиеся у вас знания .NET, вы можете поинтересоваться, почему в файле \*.aspx нет указаний на пространства имен System.Data и System. Причина в том, что все страницы \*.aspx автоматически получают доступ в ряду ключевых пространств имен, включай следующие.

• System

• System.Collections

• System.Collections.Generic

• System.Configuration

• System.IO

• System.Text

• System.Text.RegularExpressions

• Все пространства имен, связанные с System.Web

ASP.NET определяет ряд других директив, которые могут встречаться в файлах \*.aspx как до, так и после ‹%Page%› и ‹%@Import%›, но их обсуждение предполагается привести позже.

**Блок ‹script›**

В соответствии с моделью одномодульной страницы файл \*.aspx может содержать логику сценария серверной стороны, который должен выполняться на Web-сервере. Блоки программного кода, определенные для сервера, должны выполняться на сервере, поэтому для них используется атрибут runat="server". Если атрибут runat="server" не указан, среда выполнения предполагает, что соответствующий блок является сценарием *клиента,*который следует отправить с исходящим HTTP-ответом.

‹script runat="server"›

protected void btnFillData\_Click(object Sender, EventArgs e) {}

‹/script›

Сигнатура этого вспомогательного метода должна выглядеть очень знакомой. Вспомните, что при изучении Windows Forms говорилось о том, что обработчик события должен соответствовать шаблону, определенному соответствующим делегатом .NET. А когда вы хотите обработать щелчок на кнопке со стороны сервера, соответствующим делегатом является System.EventHandler, который, как вы помните, может вызвать только методы, получающие в качестве первого параметра System.Object, а в качестве второго – System.EventArgs.

**Декларация элемента ASP.NET**

Последним из рассматриваемых здесь вопросов является структура определения элементов управления Button, Label и GridView Web-формы. Подобно ASP и HTML, Web-элементы ASP.NET размещаются в контексте ‹form›. Но в этом случае открывающий дескриптор ‹form› сопровождается атрибутом runat="server". Это очень важно, поскольку тем самым дескриптор информирует среду выполнения ASP.NET о том, что перед размещением HTML-кода в потоке ответа соответствующие элементы ASP.NET должны получить возможность обновить свое HTML-представление.

‹form id="form1" runat="server"›

...

‹/form›

**Исходный код.**Файл примера SinglePageModel размещен в подкаталоге, соответствующем главе 23.

**Модель страницы с внешним кодом поддержки**

Чтобы продемонстрировать возможности модели страницы с внешним кодом поддержки, мы воссоздадим предыдущий пример, используя шаблон Web-узла Visual Studio 2005 (при этом важно понимать, что для создания страниц с внешним кодом поддержки использовать Visual Studio 2005 совсем не обязательно). Выбрав File→New→Web Site из меню, укажите шаблон ASP.NET Web Site (рис. 23.11).

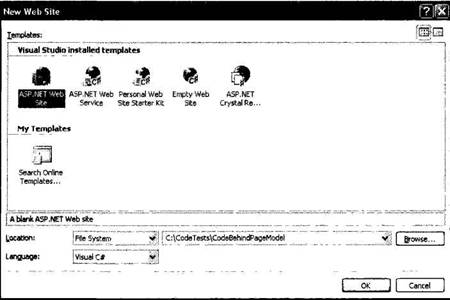


Рис. 23.11. Шаблон ASP.NET Web Site в Visual Studio 2005

На рис 23.11 обратите внимание на то, что вы можете сразу указать место расположения нового узла. При выборе File System ваши файлы будут размещены в пределах одного локального каталога, и страницы будут обслуживаться с помощью WebDev.WebServer.exe. Если выбрать FTP или HTTP, узел будет обслуживаться в рамках виртуального каталога, поддерживаемого IIS. Для нашего примера нет никакой разницы, какую из возможностей вы выберете, но для простоты давайте выберем File System.

**Замечание.**При создании Web-узла ASP.NET В Visual Studio 2005 соответствующий файл решения, (\*.sln) по умолчанию размещается в лапке Мои документы\Visual Studio 2005\Projects. Файлы содержимого узла (такие как, например, \*.аspx) будут находиться в указанном локальном каталоге или (при использовании IIS) в физическом файле, отображающемся в виртуальный каталог.

Снова используйте окно проектирования для построения пользовательского интерфейса, состоящего из Label, Button и GridView, и используйте окно свойств для изменения настроек. Теперь щелкните на кнопке Source внизу окна, чтобы увидеть окно программного вода, и вы увидите ожидаемые дескрипторы ‹asp› и ‹/asp›. Также обратите внимание на то, что директива ‹%@Page%› в данном случае имеет два новых атрибута.

‹\*@Page Language="C#" AutoEventWireup="true" **CodeFile="Default.aspx.cs" Inherits="\_Default" %›**

Атрибут CodeFile используется для указания связанного внешнего файла, содержащего программную логику страницы. По умолчанию имена файлов с внешним кодом поддержки образуются путем добавления суффикса.сs к имени файла \*.aspx (скажем, в нашем примере это Default.aspx.cs). Если заглянуть в окно Solution Explorer, вы увидите файл с внешним кодом поддержки в рамках узла Web-формы (рис. 23.12).

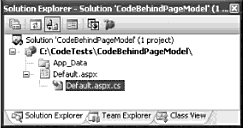


Рис. 23.12. Файл с внешним кодом поддержки, ассоциированный с файлом \*.aspx.

**Замечание.** Атрибут Codebehind, предлагавшийся в ASP.NET 1.x, в рамках директивы ‹!@Page%› больше не поддерживается.

Кроме ряда операторов using для указания связанных с Web пространств имен, ваш файл внешнего программного кода определяет класс с модификатором partial, производный от System.Web.UI.Page. Обратите внимание на то, что имя этого класса (\_Dеfault) идентично значению атрибута inherits, указанного в рамках директивы ‹%@Page%› (подробнее о Page\_Load() мы поговорим в этой главе немного позже).

public partial class **\_Default**: System.Web.UI.Page {

 protected void Page\_Load(object sender, EventArgs e) {

 }

}

Обработайте событие Click для типа Button (снова аналогично приложениям Windows Forms). Как и раньше, в определение Button будет добавлен атрибут OnClick. Однако теперь обработчик события сервера уже не размещается в контексте ‹script› файла \*.aspx, а оказывается методом типа класса \_Default. В завершение построения примера добавьте оператор using для System.Data.SqlClient в файл с внешним кодом поддержки и реализуйте программу обработки в соответствии с предыдущей программной логикой ADO.NET.

protected void btnFillGrid\_Click(object sender, EventArgs e) {

 SqlConnection sqlConn = new SqlConnection("Data Source=.;Initial Catalog=Cars;UID=sa;PWD");

 sqlConn.Open();

 SqlCommand cmd = new SqlCommand("Select \* From Inventory", sqlConn);

 carsGridView.DataSource = cmd.ExecuteReader();

 carsGridView.DataBind();

 sqlConn.Close();

}

Если при создании проекта вы выбрали вариант File System, то при выполнении Web-приложения автоматически cтартует WebDev.WebServer.exe (очевидно, что при выборе IIS этого не будет). В любом случае используемый по умолчанию браузер должен отобразить содержимое страницы.

**Отладка и трассировка страниц ASP.NET**

Вообще говоря, при создании Web-проекта ASP.NET вы можете использовать те же средства отладки, что и при создании любого другого проекта в Visual Studio 2005. Так, вы можете устанавливать контрольные точки в файле внешнего кода поддержки (и в блоках script файла \*.aspx), запускать сеанс отладки (по умолчанию для этого используется клавиша ‹F5›) и использовать режим пошагового выполнения программного кода.

Но, чтобы выполнять отладку Web-приложения ASP.NET, ваш узел должен содержать правильно скомпонованный файл Web.config. В главе 24 структура файлов Web.config рассматривается подробнее, но, по существу, эти XML-файлы служат той же цели, что и файл app.config выполняемого компоновочного блока. Если ваш проект еще не содержит файла Web.config, Visual Studio 2005 это обнаружит и добавит такой файл в ваш проект. Соответствующим элементом является ‹compilation›.

‹configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0"›

 …

 ‹system.web›

**‹compilation debug="true"/›**

 ‹/system.web›

‹/configuration›

Вы также можете разрешить поддержку трассировки для файла \*.aspx, установив для атрибута Trace значение true (истина) в рамках директивы ‹%@Page%›.

‹%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="\_Default" **Trace="true" %›**

В результате генерируемый HTML-код будет содержать многочисленные подробности, касающиеся предыдущего цикла запроса/ответа HTTP (переменные сервера, сеанса, приложения и т.д.). Чтобы добавить к ним свои собственные сообщения трассировки, можете использовать свойство Trace типа System.Web.UI.Page. В любое время, когда вы пожелаете записать пользовательское сообщение (из блока сценария или файла исходного вода C#), просто вызовите метод Write().

protected void btnFillGrid\_Click(object Sender, EventArgs e) {

 …

**// Генерирование пользовательского сообщения трассировки.**

 Trace.Write("Моя категория", "Конец заполнения таблицы");

}

Если теперь запустить проект и направить вторичное обращение Web-серверу, вы увидите свою пользовательскую категорию и пользовательское сообщение ближе к концу раздела трассировки перед разделом Control Tree (рис. 23.13).



Рис. 23.13. Запись пользовательских сообщений трассировки

**Исходный код.**Файл примера CodeBehindPageModel размещен в подкаталоге, соответствующем главе 23.

**Структура каталогов Web-узла ASP.NET**

Если у вас есть опыт создания Web-приложений с использованием ASP.NET 1.x, для вас можете показаться весьма удивительным то, что множество привычных для вас файлов (Web.config, Global.asax, AssemblyInfo.cs и т.д.) новый Web-узел не включает. Кроме того, шаблон Web Site предлагает папку App\_Data, но, кажется, в окне Solution Explorer отсутствует папка References.

Прежде всего, следует подчеркнуть, что файлы Web.config И Global.asax, конечно же, в ASP.NET 2.0 поддерживаются, но вам нужно явно добавить их в проект, выбрав WebSite→Add New Item из меню.

В главе 24 будет рассмотрена роль этих двух файлов, поэтому пока что о деталях не беспокойтесь. Знайте также, что вы можете добавить для Web-узла любое число ссылок на внешние компоновочные блоки .NET с помощью выбора WebSite→Add Reference из меню (при этом, как мы позже убедимся, результат будет немного отличаться от интуитивно ожидаемого).

Другим существенным отличием новой схемы Web-приложений является то, что в Visual Studio 2005 Web-узлы могут содержать целый ряд подкаталогов со специальными именами, имеющими специальные значения в среде выполнения ASP.NET. Эти "специальные подкаталоги" описаны в табл. 23.3.

**Таблица 23.3.**Специальные подкаталоги ASP.NET 2.0

|  |  |
| --- | --- |
| **Подкаталог** | **Описание** |
| App\_Browsers | Папка для файлов определений, которые используются для идентификации браузеров и выявления их возможностей |
| App\_Code | Папка для исходного кода компонентов или классов, которые вы хотите компилировать, как часть вашего приложения. Программный код из этого подкаталога компилируется при запросе страниц и автоматически будет доступен вашему приложению |
| App\_Data | Папка для хранения файлов \*.mdb Access, файлов \*.mdf SQL Express, XML-файлов и других наборов данных |
| App\_GlobalResources | Папка для файлов \*.resx, которые доступны из программного кода приложения |
| App\_LocalResources | Папка для файлов \*.resx, которые привязаны к конкретной странице |
| App\_Themes | Папка с набором файлов, определяющих внешний вид Web-страницы и элементов управления ASP.NET |
| App\_WebReferences | Папка для классов агентов, схем и других файлов, связанных с использованием Web-сервисов в приложении |
| Bin | Папка для скомпилированных приватных компоновочных блоков (файлы \*.dll). На компоновочные блоки из папки Bin приложение ссылается автоматически |

Добавить любую из этих подпапок в Web-приложение можно явно, выбрав WebSite→Add Folder из меню. Но во многих случаях это сделает сама среда разработки, как только вы "естественным образом" добавите соответствующий файл (например, при добавлении в систему узла нового файла C#, автоматически в структуру каталогов добавляется папка App\_Code, если она в этот момент не существует).

**Роль папки Bin**

Позже вы увидите, что Web-страницы ASP.NET в конечном счете компилируются в компоновочный блок .NET. Поэтому не должно быть неожиданностью то, что Web-узлы могут ссылаться на любое число приватных или общедоступных компоновочных блоков, В ASP.NET 2.0 метод указания внешних компоновочных блоков, необходимых для данного узла, в корне отличается от того, что предлагалось в рамках ASP.NET 1.x. Причина такого изменения в том, что теперь в Visual Studio 2005 Web-узлы трактуются в *беспроектной форме.*

Хотя шаблон Web Site и генерирует файл \*.sln, с помощью которого можно загрузить файлы \*.aspx в среду разработки, связанного с ним файла \*.csproj не существует. Вы, возможно, знаете, что проект Web-приложения ASP.NET 1.x записывал информацию обо всех внешних компоновочных блоках в файл \*.csproj. Этот факт порождает резонный вопрос: "Где хранится информация о внешних компоновочных блоках в ASP.NET 2.0?"

Когда вы ссылаетесь на приватный компоновочный блок, Visual Studio 2005 автоматически создает каталог Bin в структуре каталогов приложения, чтобы сохранить там локальную копию двоичного файла. При использовании вашим программным кодом типов из соответствующих библиотек программного кода они автоматически загружаются по первому запросу. Для проверки активизируйте меню WebSite→Add Reference и выберите любой (но не строго именованный) файл \*.dll из тех, которые вы создали в процессе изучения текста этой книги, и вы обнаружите, что в окне Solution Explorer отображается папка Bin (рис. 23.14).

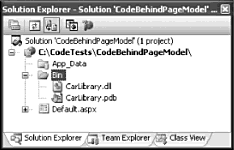


Рис. 23.14. Папка Bin содержит копии всех приватных компоновочных блоков, на которые ссылается приложение

Если же вы ссылаетесь на общедоступный компоновочный блок, Visual Studio 2006 автоматически добавляет в текущее Web-решение файл web.config (если его еще нет) и записывает внешнюю ссылку в рамках элемента ‹assemblies›. Так, если снова активизировать меню Site→Add Reference, но на этот раз выбрать общедоступный компоновочный блек (например. System.Drawing.dll), то вы обнаружите, что ваш файл Web.config примет следующий вид.

‹?xml version="1.0"?›

‹configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0"›

 ‹appSettings*/›*

 ‹connectionStrings/›

 ‹system.web›

  ‹compilation debug="false"›

**‹assemblies›**

**‹add assembly="System.Drawing, Version=2.0.0.0, Culture=neutral, PublicKeyToken=B03F5F7F11D50A3A"/›**

**‹/assemblies›**

  ‹/compilation›

  ‹authentication mode="Windows"/›

 ‹/system.web›

‹/configuration›

Как видите, каждый компоновочный блок описывается с помощью той же информации, которая требуется для динамической загрузки через метод Assembly.Load() (см. главу 12).

**Роль папки App\_Code**

Папка App\_Code используется для хранения файлов исходного кода, которые не привязаны непосредственно к конкретной Web-странице (как файлы с внешним кодом поддержки), но которые все равно должны компилироваться для использования вашим Web-узлом. Программный код из папки App\_Code будет автоматически компилироваться в фоновом режиме по мере необходимости. После этого соответствующий компоновочный блок становится доступным любому другому программному коду Web-узла. В этом смысле папка App\_Code во многом подобна папке Bin, за исключением того, что здесь вы можете сохранить исходный код вместо скомпилированного программного кода. Главным преимуществом такого подхода является то, что оказывается возможным определить пользовательские типы для Web-приложения без необходимости компилировать их независимо.

Одна папка App\_Code может содержать файлы программного кода, созданные на разных языках. В среде выполнения подходящий компилятор сгенерирует нужный компоновочный блок. Если же вы предпочитаете хранить такие файлы программного кода раздельно, можете определить множество подкаталогов для хранения файлов с управляемым программным кодом разного типа (\*.cs, \*.vb и т.д.).

Для примера предположим, что вы добавили в корневой каталог приложения Web-узла папку App\_Code, содержащую две подпапки (MyCSharpCode и MyVbNetCode), которые содержат файлы, написанные на соответствующих языках. После этого вы можете создать файл Web.config, который указывает на эти подпапки с помощью элемента ‹codeSubDirectories›.

‹?xml version="1.0"?›

‹configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0"›

 ‹appSettings/›

 ‹connectionStrings/›

 ‹system.web›

  ‹compilation debug="false"›

   ‹assemblies›

    ‹add assembly="System.Drawing, Version=2.0.0.0, Culture=neutral, PublicKeyToken=B03F5F7F11D50A3A"/›

   ‹/assemblies›

**‹codeSubDirectories›**

**‹add directoryName="MyCSharpCode" /›**

**‹add directoryName="MyVbNetCode" /›**

**‹/codeSubDirectories›**

  ‹/compilation›

  ‹authentication mode="Windows"/›

 ‹/system.web›

‹/configuration›

**Замечание.**Папка App\_Code часто используется и для хранения файлов, которые не являются файлами c программным кодом на конкретном языке, но тоже оказываются необходимыми (например, файлы \*.xsd, \*.wsdl и т.д).

**Цикл компиляции страницы ASP.NET 2.0**

Независимо от того, какую модель страницы вы использовали (одномодульную страницу или страницу с внешним кодом поддержки), ваши файлы \*.aspx (как и любые связанные файлы с кодом поддержки) динамически компилируются в действительный компоновочный блок .NET. Этот компоновочный блок затем обрабатывается в рамках рабочего процесса ASP.NET (aspnet\_wp.exe) в пределах собственного домена приложения (для получения более подробной информации о доменах приложений см. главу 13). Однако метод компиляции компоновочного блока Web-узла в ASP.NET 2.0 оказывается совершенно иным.

**Цикл компиляции одномодульных страниц**

При использовании модели одномодульной страницы, HTML-разметка, блоки ‹script› и определения Web-элементов управления динамически компилируются в тип класса, производный от System.Web.UI.Page.

Имя этого класса получается из имени файла \*.aspx с помощью присоединения суффикса \_аspx к имени файла (например, страница MyPage.aspx порождает тип класса с именем MyPage\_aspx). На рис. 23.15 показана общая схема соответствующего процесса.

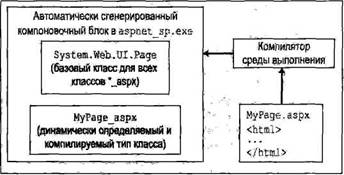


Рис. 23.15. Модель компиляции одномодульных страниц

Этот динамически компилируемый компоновочный блок устанавливается в определенный средой выполнения подкаталог в папке ‹%windir%›Microsoft.NET\ Framework\v2.0.50215\Temporary ASP.NET Files\root. Имя пути после \root зависит от целого ряда факторов (хеш-кода и т.п.). но в конце концов там можно найти соответствующие файлы \*.dll (и файлы поддержки). На рис. 23.16 показан пример одного такого компоновочного блока.

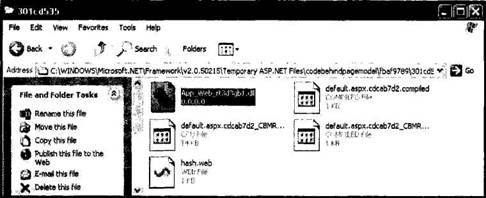


Рис. 23.16. Автоматически сгенерированный компоновочный блок ASP.NET

**Цикл компиляции многомодульных страниц**

Процесс компиляции страницы, построенной по модели с внешним кодом поддержки, подобен процессу компиляции одномодульной страницы. Однако получающийся при этом тип, производный от System.Web.UI.Page, компонуется из трех файлов (да, именно из трех, а не из ожидаемых двух).

Взглянув на предыдущий пример CodeBehindPageModel, вспомните о том, что файл Default.aspx связывается с парциальным классом \_Default, размещенным в файле внешнего кода поддержки. Если вы имеете опыт работы с ASP.NET 1.x, то можете спросить, что же при этом происходит с описаниями членов-переменных для различных Web-элементов управления и с программным кодом в пределах InitializeComponent(), в частности с программной логикой обработки событий. В ASP.NET 2.0 все это собирается в третьем "файле", генерируемом в памяти. Фактически это не совсем файл, а представление парциального класса в памяти (рис. 23.17).

В рамках этой модели объявленные в файле \*.aspx Web-элементы управления используются для построения дополнительного парциального класса, определяющего все члены-переменные интерфейса пользователя и программную логику конфигурации, которые в ASP.NET 1.x обычно находились в пределах метода InitializeComponent(), а в данном случае остаются для нас невидимыми. Этот парциальный класс в процессе компиляции объединяется с файлом внешнего кода поддержки, чтобы в результате получился *базовый класс*генерируемого типа класса \_aspx (в модели компиляции одномодульной страницы генерируемый файл \_aspx получается непосредственно из System.Web.UI.Page).

В любом случае после создания компоновочного блока в ответ на исходный HTTP-запрос этот компоновочный блок будет использоваться многократно для всех последующих запросов, не требуя перекомпиляции. Вот почему первый запрос страницы \*.aspx может занимать много времени, а последующие обращения к той же страницы оказываются намного быстрее.



Рис. 23.17. Модель компиляции многомодульных страниц

**Замечание.**В ASP.NET 2.0 теперь можно выполнить предкомпиляцию всех (или некоторого подмножества) страниц Web-узла с помощью специального инструмента командной строки aspnet\_compiler.exe. Более конкретная информация по этому вопросу имеется в документации .NET Framework 2.0 SDK.

**Цепочка наследования типа Page**

Как вы только что убедились, готовый генерируемый класс, представляющий файл \*.aspx, получается из System.Web.UI.Page. Подобно любому базовому классу, этот тип обеспечивает полиморфный интерфейс всем производным типам. Однако тип Page является не единственным членом в иерархии наследования. Если найти тип Page (в пределах компоновочного блока System.Web.dll) в окне обозреватели объектов Visual Studio 2005, то вы увидите, что этот тип "принадлежит" типу TemplateControl, который, в свою очередь, "принадлежит" Control, а последний "принадлежит" Object (рис. 23.18).

Вы должны догадываться, что каждый из этих базовых классов вносит в файл \*.aspx свой "немалый вклад" в отношении функциональных возможностей. Для большинства проектов вы будете использовать члены, определенные в рамках родительских классов Page и Control. Вообще говоря, функциональные возможности, приобретенные от класса System.Web.UI.TemplateControl, могут представлять для вас интерес только при построении пользовательских элементов управления Web Form и при взаимодействии с процессом визуализации. С этими оговорками давайте рассмотрим роль типа Page.

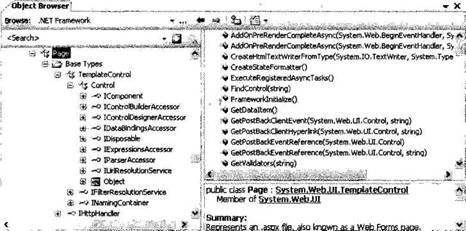


Рис. 23.18. Происхождение страницы ASP.NET

**Тип System.Web.UI.Page**

Первым интересующим нас родительским классом является сам класс Page. Ниже описаны его многочисленные свойства, обеспечивающие возможность взаимодействия с различными Web-примитивами, такими как переменные приложения и сеанса, запросы и ответы HTTP, темы и т.д. Описания некоторых их этих свойств приводятся в табл. 23.4.

**Таблица 23.4.**Свойства типа Page

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Application | Позволяет взаимодействовать с переменными приложения для текущего Web-узла |
| Cache | Позволяет взаимодействовать с объектом кэша для текущего Web-узла |
| ClientTarget | Позволяет указать способ визуализации для данной страницы в зависимости от запрашивающего браузера |
| IsPostBack | Получает значение, являющееся индикатором загрузки страницы в ответ на вторичный запрос клиента (в отличие от первичной загрузки страницы) |
| MasterPageFile | Создает шаблон страницы для текущей страницы |
| Request | Обеспечивает доступ к текущему HTTP-запросу |
| Response | Позволяет взаимодействовать с исходящим HTTP-ответом |
| Server | Обеспечивает доступ к объекту HttpServerUtility, содержащему различные вспомогательные функции сервера |
| Session | Позволяет взаимодействовать с сеансовыми данными для текущего вызывающего объекта |
| Theme | Получает или устанавливает имя темы, используемой для текущей страницы |
| Trace | Обеспечивает доступ к объекту TraceContext, позволяющему записывать пользовательские сообщения в ходе сеанса отладки |

**Взаимодействие с поступающим HTTP-запросом**

Вы уже видели выше, что основной поток Web-сеанса начинается с регистрации клиента, ввода пользовательской информации и щелчка на кнопке Отправить, в результате чего данные HTML-формы направляются Web-странице для обработки. В большинстве случаев открывающий дескриптор form содержит атрибуты action и method, указывающие файл на Web-сервере, который должен обеспечить данные различным HTML-элементам, и метод пересылки этих данных (GET или POST).

‹form name="defaultPage" id="defaultPage" action="http://localhost/Cars/ClassicAspPage.asp" method = "GET"›

‹/form›

В отличие от классической технологии ASP, в рамках ASP.NET объект с именем Request не поддерживается. Однако все страницы ASP.NET наследуют свойство System.Web.UI.Page.Request, обеспечивающее доступ к экземпляру типа класса HttpRequest. В табл. 23.5 предлагаются описания некоторых базовых членов указанного типа, и не удивительно, что эти члены предлагают возможности, аналогичные возможностям членов, присутствующих в уже устаревшем объекте Request классической модели ASP.

**Таблица 23.5.**Члены типа HttpRequest

|  |  |
| --- | --- |
| **Член** | **Описание** |
| ApplicationPath | Получает путь к виртуальному каталогу приложения ASP.NET на сервере |
| Browser | Обеспечивает информацию о возможностях браузера клиента |
| Cookies | Получает коллекцию файлов cookie, отправленных браузером клиента |
| FilePath | Указывает виртуальный путь текущего запроса |
| Form | Получает коллекцию переменных формы |
| Headers | Получает коллекцию HTTP-заголовков |
| HttpMethod | Указывает метод передачи HTTP-данных, используемый клиентом (GET, POST) |
| IsSecureConnection | Индикатор защищенности HTTP-соединения (т.е. использования HTTPS) |
| QueryString | Получает коллекцию строковых переменных HTTP-запроса |
| RawUrl | Получает "сырой" URL текущего запроса |
| RequestType | Указывает метод передачи HTTP-данных, используемый клиентом (GET, POST) |
| ServerVariables | Получает коллекцию переменных Web-сервера |
| UserHostAddress | Получает IP-адрес хоста удаленного клиента |
| UserHostName | Получает DNS-имя удаленного клиента |

В дополнение к этим свойствам тип HttpRequest предлагает ряд полезных методов, включая следующие.

• MapPath(). Отображает виртуальный путь запрошенного адреса URL в физический путь на сервере для текущего запроса.

• SaveAs(). Сохраняет информацию текущего HTTP-запроса в файл на Web-сервере (что может оказаться полезным при отладке).

• ValidateInput(). Если с помощью атрибута Validate соответствующей директивы страницы разрешена возможность контроля данных, то этот метод может вызываться для проверки всех вводимых пользователем данных (включая данные cookie) на случай выявления потенциально опасных вариантов ввода (из предусмотренного списка таких вариантов).

**Получение статистики браузера**

Первый интересным элементом типа HttpRequest является свойство Browser, обеспечивающее доступ к базовому объекту HttpBrowserCapabilities. Объект HttpBrowserCapabilities, в свою очередь, предлагает множество членов, которые позволяют программно исследовать статистику браузера, отправившего поступивший HTTP-запрос.

Создайте новый Web-узел ASP.NET с именем FunWithPageMembers. Нашим первым заданием будет построение пользовательского интерфейса, позволяющего при щелчке пользователя на Web-элементе управления Button увидеть различную информацию о вызывающем браузере. Эта информация будет генерироваться динамически и присваиваться типу Label (с именем lblOutput). Обработчик события Click для Button будет таким.

protected void btnGetBrowserStats\_Click(object sender, System.EventArgs e) {

 string theInfo = "";

 theInfo += String.Format ("‹li›Это клиент AOL? {0}", Request**.Browser.AOL**);

 theInfo += String.Format("‹li›Поддерживает ли клиент ActiveX? {0}", Request**.Browser.ActiveXControls**);

 theInfo += String.Format("‹li›Это клиент Beta? {0}", Request**.Browser.Beta**);

 theInfo += String.Format("‹li›Поддерживает ли клиент Java? {0}", Request.**Browser.JavaApplets**);

 theInfo += String.Format("‹li›Поддерживает ли клиент cookie? {0}", Request**.Browser.Cookies**);

 theInfo += String.Format("‹li›Поддерживает ли клиент VBScript? {0}", Request.**Browser.VBScript**);

 lblOutput.Text = theInfo;

}

Здесь проверяется целый ряд возможностей браузера. Как вы можете догадываться, очень важно выяснить возможность поддержки браузером элементов управления ActiveX, апплетов Java и VBScript клиента. Если вызывающий браузер не поддерживает какую-то из Web-технологий, ваша страница \*.aspx должна быть готова выполнить альтернативный план действий.

**Доступ к поступающим данным формы**

Другими элементами типа HttpResponse являются свойства Form и QueryString. Эти два свойства функционируют аналогично классическому варианту ASP и позволяют анализировать поступающие данные формы, используя пары имен и значений. Вспомните из нашего предыдущего обсуждения классической технологии ASP о том, что при отправке HTTP-данных с помощью GET данные формы будут доступны через свойство QueryString, тогда как для доступа к данным, представленным с помощью POST, используется свойство Form.

Для доступа к данным формы клиента на Web-сервере, конечно, можно использовать свойства HttpRequest.Form и HttpRequest.QueryString, но этот устаревший подход (в большинстве случаев) не является необходимым. Ввиду того, что ASP.NET предлагает свои собственные Web-элементы управления серверной стороны, у вас есть возможность обращаться с HTML-элементами интерфейса, как с настоящими объектами. Таким образом, вместо получения значения текстового блока в варианте

protected void btnGetFormData\_Click(object sender, EventArgs e) {

**// Получение значения для элемента с ID=txtFirstName.**

 string firstName = Request.Form["txtFirstName"];

}

вы можете напрямую запросить свойство Text серверного элемента управления.

protested void btnGetFormData\_Click(object sender, EventArgs e) {

**// Получение значения для элемента с ID=txtFirstName.**

 string firstName = txtFirstName.Text;

}

Этот подход не только соответствует строгим принципам ООП, но при этом вообще не приходится заботиться о том, как представляются данные формы (GET или POST). К тому же непосредственная работа с элементом управления гораздо больше соответствует требованиям типовой безопасности, поскольку здесь возможные ошибки ввода будут выявлены уже на этапе компиляции, а не в среде выполнения. Конечно, это не значит, что вам в ASP.NET вообще никогда не придется использовать свойства Form и QueryString, но необходимость в их использовании существенно уменьшится.

**Свойство IsPostBack**

Еще одним очень важным членом HttpRequest является свойство IsPostBack. Напомним, что "postback" обозначает вторичное обращение к конкретной Web-странице в ходе одного сеанса связи с сервером. С учетом этого должно быть понятно, что свойство IsPostBack возвращает true (истина), если текущий HTTP-запрос отправлен уже зарегистрированным настоящий момент пользователем, и false (ложь), если это первое взаимодействие пользователя со страницей.

Обычно необходимость в определении того, что текущий HTTP-запрос является вторичным, возникает тогда, когда некоторый блок программного кода должен выполняться только при первом обращении пользователя к странице. Например, при первом доступе пользователя к файлу \*.aspx вы можете заполнить некоторый объект DataSet ADO.NET и поместить этот объект в кэш для использования в дальнейшем. Когда вызывающая сторона снова обратится к той же странице, вы можете избежать необходимости нового обращения к базе данных (конечно, некоторые страницы могут требовать, чтобы DataSet обновлялся при каждом запросе, но это уже другая проблема).

protected void Page\_Load(objeet sender, EventArgs e) {

**// DataSet заполняется только при первом обращении**

**// пользователя к данной странице.**

 if (!IsPostBack) {

**// Заполнение DataSet и отправка в кэш!**

 }

**// Использование DataSet из кэша.**

}

**Взаимодействие с исходящим HTTP-ответом**

Теперь вы понимаете, как тип Page взаимодействует с поступающим HTTP-за-просом, и следующим шагом должно быть выяснение того, как реализуется взаимодействие с исходящим HTTP-ответом. В ASP.NET свойство Response класса Page обеспечивает доступ к экземпляру типа HttpResponse. Этот тип определяет ряд свойств, позволяющих сформировать HTTP-ответ, отправляемый обратно браузеру клиента. Описания базовых свойств этого типа предлагаются в табл. 23.6.

**Таблица 23.6.**Свойства типа HttpResponse

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Cache | Возвращает семантику кэширования Web-страницы (например, время  ожидания, параметры конфиденциальности, различные описания) |
| ContentEncoding | Читает или устанавливает набор символов выходного потока HTTP |
| ContentType | Читает или устанавливает MIME-тип выходного потока HTTP |
| Cookies | Получает коллекцию HttpCookie, посланную текущим запросом |
| IsClientConnected | Читает значение, являющееся индикатором продолжающегося соединения клиента с сервером |
| Output | Разрешает пользовательский вывод в поле содержимого исходящего HTTP-сообщения |
| OutputStream | Разрешает двоичный вывод в поле содержимого исходящего HTTP-сообщения |
| StatusCode | Читает или устанавливает код состояния HTTP-ответа, возвращаемого клиенту |
| StatusDescription | Читает или устанавливает строку состояния HTTP-ответа, возвращаемого клиенту |
| SuppressContent | Читает или устанавливает значение, являющееся индикатором отмены  отправки HTTP-содержимого клиенту |

Рассмотрите также описания некоторых методов типа HttpResponse, представленные в табл. 23.7.

**Таблица 23.7.**Методы типа HttpResponse

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| AddCacheDependency() | Добавляет объект в кэш приложения (см. главу 24) |
| Clear() | Удаляет все заголовки и содержимое вывода из буфера потока |
| End() | Отправляет все содержимое буфера вывода клиенту, а затем завершает  соединение для данного сокета |
| Flush() | Отправляет все содержимое буфера вывода клиенту |
| Redirect() | Выполняет перенаправление клиента по новому URL |
| Write() | Записывает значения в выходной поток HTTP-содержимого |
| WriteFile() | Записывает файл непосредственно в выходной поток HTTP-содержимого |

**Генерирование HTML-содержимого**

Пожалуй, самой известной сферой применения типа HttpResponse является запись содержимого непосредственно в выходной поток HTTP. Метод HttpResponse. Write() позволяет передать HTML-дескрипторы, или вообще любые строковые литералы. Метод HttpResponse.WriteFile() расширяет эти возможности с тем, чтобы вы могли указать имя физического файла на Web-сервере, содержащего данные, направляемые в выходной поток (это оказывается очень удобным в том случае, когда требуется отправить содержимое уже существующего файла \*.htm).

Для примера предположим, что вы добавили в свой файл \*.aspx еще один тип Button, который реализует обработчик события Click сервера так.

protected void btnHttpResponse\_Click(object sender, EventArgs e) {

 Response**.Write(**"‹b›Moe имя **:**‹/b›‹br›");

 Response**.Write**(this.ToString());

 Response**.Write**("‹br›‹br›‹b›Boт Ваш последний запрос:‹/b›‹br›");

 Response.**WriteFile(**"MyHTMLPage.htm");

}

Роль этой вспомогательной функции (которая может вызываться некоторым: обработчиком события на стороне сервера) очень проста. Единственным заслуживающим внимания моментам здесь является то, что метод HttpResponse. WriteFile() теперь отправляет содержимое файла \*.htm сервера из корневого каталога Web-узла.

Снова подчеркнем, что вы, конечно, можете использовать подход "старой школы", чтобы отображать HTML-дескрипторы и содержимое, используя метод Write(), но этот подход в рамках ASP.NET применяется гораздо реже, чем в рамках классической технологии ASP. Причина здесь (снова) в наличии серверных Web-элементов управления. Скажем, чтобы отобразить блок текстовых данных в браузере, достаточно просто присвоить подходящее значение свойству Text элемента Label.

**Перенаправление пользователей**

Другой возможностью типа HttpResponse является перенаправление пользователя по новому адресу URL.

protected void btnSomeTraining\_Click(object sender, EventArgs e) {

 Response.Redirect("http://www.IntertechTraining.com");

}

Если этот обработчик событий вызвать с помощью вторичного обращения клиента к серверу, пользователь будет автоматически перенаправлен по указанному URL.

**Замечание.**Вызов метода HttpResponse.Redirect() всегда влечет за собой обращение к браузеру клиента. Если нужно просто передать управление файлу \*.aspx в том же виртуальном каталоге, более эффективным будет вызов метода HttpServerUtility.Transfer() (доступного через наследуемое свойство Server).

На этом мы завершим обсуждение функциональных возможностей System.Web.UI.Page. Чуть позже мы рассмотрим роль базового класса System.Web.UI.Control, однако нашим следующим заданием будет исследование цикла существования объектов, производных от Page.

**Исходный код.**Файлы примера FunWithPageMembers размещены в подкаталоге, соответствующем главе 23.

**Цикл существования Web-страницы ASP.NET**

Каждая Web-страница ASP.NET имеет свой "жизненный цикл". Когда среда выполнения ASP.NET получает входящий запрос для данного файла \*. aspx, в памяти размещается соответствующий тип, производный от System.Web.UI.Page, для создания которого используется конструктор, заданный по умолчанию. После этого среда обработки автоматически генерирует серию событий.

По умолчанию сгенерированная в Visual Studio 2005 страница с внешним кодом поддержки определяет обработчик события Load страницы.

public partial class \_Default: System.Web.UI.Page {

 protected void Page\_Load(object sender, EventArgs e) {

 }

}

Кроме события Load, тип Page может выполнять перехват любого из событий, указанных в табл. 23.8 в том порядке, в котором эти события возникают.

**Таблица 23.8.**События типа Page

|  |  |
| --- | --- |
| **Событие** | **Описание** |
| PreInit | Используется инфраструктурой .NET для размещения Web-элементов управления, применения тем, создания шаблона страницы и установки профиля пользователя. Вы можете перехватить это событие, чтобы внести изменения в соответствующий процесс |
| Init | Используется для установки свойств Web-элементов управления в предыдущее состояние с помощью вторичного запроса или просмотра данных состояния (подробнее об этом говорится в главе 24) |
| Load | Возникает тогда, когда страница и ее элементы управления полностью инициализированы, а их предыдущие значения восстановлены. С этого момента вполне безопасно начать взаимодействие с любым из Web-элементов |
| "Событие, вызвавшее вторичный запрос" | События с таким именем, конечно же, не существует. Так здесь обозначено любое событие, заставившее браузер отправить вторичный запрос Web-cер-веру (это может быть, например, щелчок на кнопке) |
| PreRender | Привязка данных и конфигурация пользовательского интерфейса завершена, и элементы управления готовы отправить свои данные в поток исходящего HTTP-ответа |
| Unload | Страница и её элементы управления завершили процесс передачи данных, и объект страницы готов к уничтожению. Взаимодействие с исходящим HTTP-ответом в этот момент породит ошибку среды выполнения. Можно выполнить захват этого события для "уборки мусора" на уровне страницы (чтобы закрыть файлы и базы данных, выполнить процедуру выхода из системы, освободить ресурсы и т.д.) |

**Замечание.**Все события типа Page работают с делегатом System.EventHandler.

**Роль атрибута AutoEventWireUp**

Чтобы обработать события для страницы, нужно добавить в блок ‹script› или файл с внешним кодом поддержки подходящий обработчик события. В отличие от ASP.NET 1.x. теперь не требуется вводить всю программную логику события вручную. Нужна только определить соответствующий метод, используя следующий шаблон.

protected Page\_nameOfTheEvent(object sender, EventArgs e)

Например, cобытие Unload можно обработать так.

public partial class \_Default: System.Web.UI.Page {

 protected void Page\_Load(object sender, EventArgs e) {

 }

 protected void Page\_Unload(object sender, EventArgs e) {

 }

}

Этот метод, как по волшебству, вызывается при выгрузке страницы (несмотря на то, что вы не применяли синтаксис событий C#), поскольку атрибут AutoEventWireUp устанавливается равным true (истина) по умолчанию в директиве ‹%@Page%› вашего файла \*.aspx.

‹%@ Page Language="C#" **AutoEventWireup="true"**CodeFile="Default.aspx.cs" Inherits="\_Default" %›

Как подсказывает имя этого атрибута, при его активизации будет создана необходимая оснастка событий в рамках автоматически генерируемого парциального класса, описанного в этой главе выше. Если установить этот атрибут равным false, не будут вызваны обработчики событий ни для Load, ни для Unload страницы Default (вы можете проверить это непосредственно, установив контрольные точки в пределах обработчиков событий Page\_Load() и Page\_Unload()).

Однако, если вы используете стандартный синтаксис событий C# для обработки событий Load и Unload, как показано ниже:

public partial class \_Default: System.Web.UI.Page {

 public \_Default() {

**// Явный перехват событий Load и Unload.**

  this.Load += new EventHandler(Page\_Load);

  this.Unload += new EventHandler(Page\_Unload);

 }

 protected void Page\_Load(object sender, EventArgs e) {

  Response.Write("Сработало событие Load!");

 }

 protected void Page\_Unload(object sender, EventArgs e) {

**// Направить данные в HTTP-ответ здесь невозможно,**

**// поэтому выполняется запись в локальный файл.**

  System.IO.File.WriteAllText(@"C:\MyLog.txt", "Выгрузка страницы.");

 }

 protected void btnPostback\_Click(object sender, EventArgs e) {

**// Здесь ничего не происходит, но это гарантирует**

**// вторичный запрос к странице.**

 }

}

то эти события будут перехвачены вашей страницей независимо от значения, заданного для AutoEventWireup.

В качестве заключительного замечания напомним, что с момента вызова события Unload вы не сможете взаимодействовать с подлежащим отправке HTTP-ответом (если вы попытаетесь вызвать члены объекта HttpResponse, среда выполнения сгенерирует соответствующее исключение). Поэтому здесь обработчик события Unload просто отправляет строку текста в файл на локальном диске C.

**Событие Error**

Еще одним событием, которое может происходить в цикле существования страницы, является событие Error, которое также работает в паре с делегатом System.EventHandler. Это событие возникает в том случае, когда метод производного от Page типа генерирует исключение, оставшееся без явной обработки. Предположим, что вы обработали событие Click для типа Button на странице, и в пределах обработчика события (здесь он называется btnGetFile\_Click) вы пытаетесь записать, содержимое локального файла в HTTP-ответ.

Также предположим, что вам *не удалось*проверить присутствие этого файла с помощью стандартной технологии структурированной обработки исключений. Если при этом вы предусмотрели обработку события Error страницы, вы получите шанс решить возникшую проблему, чтобы пользователь не увидел безобразную информацию об ошибке. Рассмотрите следующий программный код.

public partial class \_Default: System.Web.UI.Page {

 public \_Default() {

 …

**// Создание объекта для события Error.**

 this.Error += new EventHandler(\_Default\_Error);

 void \_Default\_Error(object sender, EventArgs е) {

**// Уничтожение текущего ответа, сообщение об сшибке**

**// и информирование среды выполнения о том,**

**// что ошибка обработана.**

  Response.Clear();

  Response.Write("Извините… не могу найти необходимый файл.");

  Server.ClearError();

 }

 protected void btnGetFile\_Click(object sender, EventArgs e) {

**// Попытка открыть несуществующий файл.**

**// Это порождает событие Error для данной страницы**.

  System.IO.File.ReadAllText(@"C:\IDontExist.txt");

 }

 …

}

Здесь обработчик события Error начинается с очистки всего содержимого имеющегося HTTP-ответа и вывода общего сообщения об ошибке. Чтобы получить доступ к конкретному объекту System.Exception, вы можете использовать метод HttpServerUtility.GetLastError(), доступ к которому обеспечивает унаследованное свойство Server.

void \_Default\_Error(object sender, EventArgs e) {

 Response.Clear();

 Response.Write("Извините… не могу найти необходимый файл. ‹br›");

 Response.Write(string.Format("Ошибка: ‹b›{0}‹/b›", Server.GetLastError().Message));

 Server.ClearError();

}

Наконец, отметьте, что перед выходом из этого общего обработчика ошибок с помощью свойства Server явно вызывается метод HttpServerUtility.ClearError(). Это необходимо, чтобы информировать среду выполнения о том, что проблема вами решена, и дальнейшего вмешательства системы не требуется. Если вы забудете сделать это, конечному пользователю будет предъявлено окно среды выполнения с сообщением об ошибке. На рис. 23.19 показан результат выполнения нашей процедуры обработки ошибок.



Рис. 23.19. Обработка ошибок на уровне страницы

В данный момент вы должны чувствовать себя довольно уверенно при работе с типом Page ASP.NET. Имея такую основу, вы теперь готовы перейти к выяснению роли Web-элементов управления ASP.NET.

**Исходный код**. Файлы примера PageLifeCycle размещены в подкаталоге, соответствующем главе 23.

**Природа Web-элементов управления**

Возможно, самым большим преимуществом ASP.NET является возможность компоновки пользовательского интерфейса страниц с помощью типов, определенных в пространстве имен System.Web.UI.WebControls. Соответствующие этим типам элементы управления (для которых могут использоваться названия *серверные элементы управления, Web-элементы управления,*или *элементы управления Web-формы)*оказываются чрезвычайно полезными в том, что они автоматически генерируют HTML-код, необходимый для запрашивающего браузера, и предлагают набор событий, которые может обработать Web-сервер, Каждому элементу управления ASP.NET соответствует класс из пространства имен System.Web.UI.WebControls, поэтому такой элемент управления может использоваться в рамках технологии ООП как в файле \*.aspx (в блоке ‹script›), так и в файле внешнего кода поддержки.

Вы уже видели, что при настройке Web-элемента управления в окне свойств Visual Studio 2005 ваши изменения записываются в определение этого элемента в файле \*.aspx в виде набора пар имен и значений. Например, при добавлении нового TextBox в окне проектирования файла \*.aspx и изменении свойств BorderStyle, BorderWidth, BackColor, BorderColor и Text средствами IDE открывающий дескриптор ‹asp:TextBox› может измениться так, как показано ниже.

‹asp:TextBox id=myTextBox runat="server" **BorderStyle=**"Ridge" **BorderWidth**="5px" **BackColor**="PaleGreen" **BorderColor**="DarkOliveGreen" **Text**= "Привет, Старик!"*›*

‹/asp:TextBox›

Поскольку HTML-декларация Web-элемента управления в конечном счете (в цикле динамический компиляции) становится членом-переменной из пространства имен System.Web.UI.WebControls, вы можете взаимодействовать с членами соответствующего типа в рамках блока ‹script› сервера или файла с внешним кодом поддержки страницы, например:

public partial class \_Default: System.Web.UI.Page {

 …

 protected void btnChangeTextBoxColor\_Click(object sender, EventArgs e) {

**// Изменение данных HTTP-ответа для данного элемента.**

  this.myTextBox.BackColor = System.Drawing.Color.Red;

 }

}

Все Web-элементы управления ASP.NET восходят к общему базовому классу с именем Sуstem.Web.UI.WebControls.WebControl. Класс WebControl получается из System.Web.UI.Control (который, в свою очередь, получается из System.Objeсt). Классы Control и WebControl определяют свои наборы свойств, общие для всех серверных элементов управлений. Перед тем как рассмотреть наследуемые функциональные возможности элементов управления, давайте выясним, что формально означает обработка серверных событий.

**Обработка серверных событий**

С учетом сегодняшнего состояния World Wide Web нельзя не принимать во внимание природу взаимодействия браузера и Web-сервера. В основе такого взаимодействия лежит цикл запросов и ответов HTTP в процессе выполнения которых состояния не сохраняются. И хотя серверные элементы управления ASP.NET делают все возможное, чтобы избавить разработчика от необходимости непосредственного обращения к настройкам протокола HTTP, никогда не забывайте о том, что трактовка Web в терминах управления событиями – это великолепный "фокус" CLR, далеко не эквивалентный модели управления событиями пользовательского интерфейса Windows.

Поэтому, хотя пространства имен System.Windows.Forms и System.Web.UI.WebControls определяют типы с аналогичными именами (Button, TextBox, GridView, Label и т.д.), они предлагают разные наборы событий. Например, когда пользователь помещает указатель мыши на поверхность Button Web-формы, у вас нет возможности обработать событие MouseMove на стороне сервера. И это, очевидно, разумно. (Кто обрадуется перспективе посылать вторичные запросы серверу при каждом движении мыши?)

Поэтому Web-элементы управления ASP.NET предлагают ограниченные наборы событий, результатом которых, в конечном итоге, оказывается новое обращение к Web-серверу. Для обработки событий*клиента*вы должны создать соответствующие элементы программного кода сценария JavaScript/VBScript клиента, которые будут обрабатываться механизмом обслуживания сценариев соответствующего браузера.

**Свойство AutoPostBack**

Следует также подчеркнуть то, что многие Web-элементы управления ASP.NET поддерживают свойство AutoPostBack (это очень важно для CheckBox, RadioButton и TextBox, а также для элементов управления, получаемых из абстрактного типа ListControl). По умолчанию это свойство получает значение false (ложь), что означает отключение автоматической отправки серверных событий (даже при наличии соответствующей настройки в файле внешнего кода поддержки). Во многих случаях это оказывается именно тем, что требуется. Но если вы хотите, чтобы какой-то из элементов управления обращался к обработчику события на сервере, нужно установить для AutoPostBack значение true (истина). Это может оказаться полезным тогда, когда данные одного элемента управления должны автоматически становиться данными другого в пределах одной и той же страницы.

Для примера создайте Web-узел, содержащий один элемент управления TextBox (с именем txtAutoPostback) и один ListBox (с именем lstTextBoxData). Затем обработайте событие TextChanged элемента TextBox и в серверном обработчике события добавьте в ListBox текущее значение TextBox (уследили за идеей?).

protected void txtAutoPostback\_TextChanged(object sender, EventArgs e) {

 lstTextBoxData.Items.Add(txtAutoPostback.Text);

}

Если выполнить приложение в таком виде, вы обнаружите, что в процессе ввода в TextBox ничего не происходит. Более того, ничего не произойдет и после ввода в TextBox при переходе к следующему элементу управления по нажатию клавиши табуляции. Причина в том, что свойство AutoPostBack типа TextBox по умолчанию имеет значение false. Но если установить для этого свойства значение true, как показано ниже:

‹asp:TextBox ID="txtAutoPostback" runat="server" **AutoPostBack="True"**OnTextChanged="txtAutoPostback\_TextChanged"›

‹/asp:TextBox›

то вы увидите, что при выходе из TextBox по нажатию клавиши табуляции (или при нажатии клавиши ‹Enter›), ListBox автоматически получает текущее значение из TextBox. Без сомнения, кроме случая добавления данных одного элемента управления в другой, необходимости изменения состояния свойства AutoPostBack в других случаях не возникает.

**Тип System.Web.UI.Control**

Базовый класс System.Web.UI.Control определяет различные свойства, методы, и события, которые позволяют взаимодействовать с базовыми членами Web-элемента управления (обычно не относящимися к графическому интерфейсу). В табл. 23.9 предлагаются описания некоторых таких членов.

**Таблица 23.5.**Подборка членов System.Web.UI.Control

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Controls | Свойство, получающее объект ControlCollection, представляющий дочерние элементы управления в рамках данного элемента управлений |
| DataBind() | Метод, выполняющий привязку источника данных к вызванному серверному элементу управления и всем его дочерним элементам управления |
| EnableTheming | Свойство, указывающее возможность поддержки тем для данного элемента управления |
| HasControls() | Метод для определения наличия дочерних элементов управления у данного серверного элемента управления |
| ID | Свойство, читающее или устанавливающее значение программного идентификатора для серверного элемента управления |
| Page | Свойство, получающее ссылку на экземпляр типа Page, содержащий серверный элемент управления |
| Parent | Свойство, получающее ссылку на родительский элемент управления данного серверного элемента управления в иерархии элементов управления страницы |
| SkinID | Свойство, читающее или устанавливающее параметры скиннинга элемента управления. Это дает возможность в ASP.NET 2.0 устанавливать внешний вид элемента управления динамически |
| Visible | Свойство, читающее или устанавливающее значение, указывающее необходимость обработки серверного элемента управления, как элемента пользовательского интерфейса страницы |

**Список вложенных элементов управления**

Первой из рассматриваемых здесь Особенностей System.Web.UI.Control является то, что все Web-элементы управления (это также относится и к Page) наследуют коллекцию пользовательских элементов управления (доступную с помощью свойства Controls). Во многом аналогично случаю приложений Windows Forms, в данном случае свойство Controls обеспечивает доступ к строго типизованной коллекции объектов WebControl. Подобно любой коллекции .NET, вы имеете возможность динамически добавлять и удалять элементы этой коллекции в среде выполнения.

Хотя добавлять Web-элементы управления в Page-тип можно и непосредственно, намного проще (и безопаснее) использовать для этого элемент управления Panel. Класс System.Web.UI.WebControls.Panel представляет контейнер элементов управления, который может быть видимым или невидимым для конечного пользователя (в зависимости от значений свойств Visible и BorderStyle).

Для примера создайте новый Web-узел с названием DynamicCtrls. В окне проектирования Web-страницы Visual Studio 2005 добавьте тип Panel (назначив ему имя myPanel), содержащий элементы TextBox, Button и HyperLink с произвольными именами (учтите, что режим проектирования требует, чтобы при перетаскивании внутренние элементы помещались в зону интерфейса типа Panel). В результате элемент ‹form› вашего файла \*.aspx должен принять следующий вид.

**‹**asp:**Panel**ID="myPanel" runat="server" Height="50px" Width="125px"›

**‹**asp:**TextBox**ID="TextBox1" runat="server"›‹/asp:TextBox›‹br /›

**‹**asp:**Button**ID="Button1" runat="server" Text="Кнопка" /›‹br /›

 ‹asp:**HyperLink**ID="HyperLink1" runat="server"›Гиперссылка‹/asp:HyperLink›

‹/asp:**Panel**›

Затем разместите элемент Label (с названием lblControlInfo) вне контекста Panel, чтобы отображать соответствующий вывод. Учтите в Page\_Load() то, что мы хотим получить список всех элементов управления, содержащихся в Panel, и присвоить полученные результаты типу Label.

public partial class \_Default: System.Web.UI.Page {

 protected void Page\_Load(object sender, EventArgs e) {

  ListControlsInPanel();

 }

 private void ListControlsInPanel() {

  string theInfo;

  theInfo = String.Format("Присутствие элементов: {0}‹br›", myPanel.**HasControls**());

  foreach (Control с in myPanel**.Controls)**{

   if (c.GetType() != typeof(System.Web.UI.LiteralControl)) {

    theInfo += "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*‹br›";

    theInfo += String.Format("Name = {0}‹br›", с.ToString());

    theInfo += String.Format("ID = {0}‹br›", c.ID);

    theInfo += String.Format("Visible = {0}‹br›", c.Visible);

    theInfo += String.Format("ViewState = {0}‹br›", c.EnableViewState);

   }

  }

  lblControlInfo.Text = theInfo;

 }

}

Здесь выполняется цикл по всем типам WebControl, поддерживаемым в Panel, и осуществляется проверка того, что текущий тип не является типом System.Web.UI.LiteralControl. Этот тип используется для представления буквальных HTML-дескрипторов и содержимого (например, ‹br›, текстовых литералов и т.д.). Если вы не выполните такой проверки, вы с удивлением можете обнаружить в контексте Panel целых семь типов (для указанного выше определения \*.aspx). В предположений о том, что тип не является буквальным HTML-содержимым, выводится определенная статистическая информация, Пример такого вывода показан на рис. 23.20.

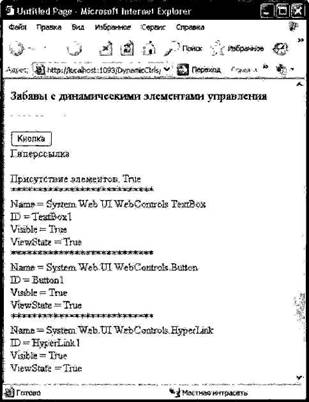


Рис. 23.20. Перечень вложенных элементов

**Динамическое добавление (и удаление) элементов управления**

Но что делать, если нужно изменить содержимое Panel в среде выполнения? Соответствующий процесс должен показаться вам очень знакомым, если вы внимательно прочитали материал книги, посвященный работе с Windows Forms. Давайте добавим в текущую страницу кнопку (с названием btnAddWidgets), которая будет динамически добавлять в Panel пять новых типов TextBox, и еще одну кнопку, которая будет выполнять очистку Panel от всех элементов управления. Обработчики событий Click для этих кнопок приведены ниже.

protected void btnAddWidgets\_Click(object sender, EventArgs e) {

 for (int i = 0; i ‹ 5; i++) {

**// Назначение имени, чтобы позже получить соответствующее**

**// текстовое значение с помощью метода**

**// HttpRequest.QueryString()**.

  TextBox t = new TextBox();

  t.ID = string.Format("newTextBox{0}", i);

  myPanel.Controls.Add(t);

  ListControlsInPanel();

 }

}

protected void btnRemovePanelItems\_Click(object sender, EventArgs e) {

 myPanel.Controls.Clear();

 ListControlsInPanel();

}

Обратите внимание на то, что каждому TextBox назначается уникальное значение ID (newTextBox1, newTextBox2 и т.д.), чтобы можно было программными средствами получить содержащийся в этих элементах текст, используя коллекцию HttpRequest.Form (как будет показано чуть позже).

Чтобы получить значения этих динамически генерируемых типов TextBox, добавьте в пользовательский интерфейс еще один тип Button и тип Label. В пределах обработчика события Click для Button реализуйте цикл по всем элементам, содержащимся в рамках типа HttpRequest.NameValueCollection (доступного с помощью HttpRequest.Form), добавляя полученную текстовую информацию к локальному типу System.String. По завершении обработки коллекции назначьте эту строку свойству Text нового элемента Label с именем lblTextBoxText.

protected void btnGetTextBoxValues\_Click(object sender, System.EventArgs e) {

 string textBoxValues = "";

 for(int i = 0; i ‹ Request.Form.Count; i++) {

  textBoxValues += string.Format("‹li›{0}‹/li›‹br›", Request.Form[i]);

 }

 lblTextBoxText.Text = textBoxValues;

}

Запустив приложение, вы сможете увидеть как содержимое текстовых блоков, так и довольно длинные ("нечитаемые") строки. Такие строки отражают *визуальное состояние*элементов на странице и будут рассматриваться позже, в следующей главе. Также вы заметите, что после обработки запроса новые текстовые окна исчезают. Причина опять кроется в природе HTTP – этот протокол не обеспечивает сохранения состояния. Чтобы динамически созданные типы TextBox сохранялись после вторичных запросов, вы должны сохранить состояния этих объектов, используя соответствующие приемы программирования ASP.NET (эти вопросы также рассматриваются в следующей главе).

**Исходный код.**Файлы примера DynamicCtrls размещены в подкаталоге, соответствующем главе 23.

**Основные члены типа System.Web.Ul.WebControls.WebControl**

Можно сказать, что тип Control предлагает возможности поведения, не относящиеся к графическому интерфейсу. С другой стороны, базовый класс WebControl обеспечивает полиморфный графический интерфейс для всех Web-элементов поведения, как показано в табл. 23.10.

**Таблица 23.10.**Свойства базового класса WebControl

|  |  |
| --- | --- |
| **Свойства** | **Описание** |
| BackColor | Читает или устанавливает цвет фона Web-элемента управления |
| BorderColor | Читает или устанавливает цвет границы Web-элемента управления |
| BorderStyle | Читает или устанавливает стиль границы Web-элемента управления |
| BorderWidth | Читает или устанавливает ширину границы Web-элемента управления |
| Enabled | Читает или устанавливает значение, являющееся индикатором доступности Web-элемента управления |
| СssСlass | Позволяет назначить Web-элементу управления класс, определенный в рамках CSS (Cascading Style Sheet – каскадная таблица стилей) |
| Font | Читает информацию о шрифте для Web-элемента управлений |
| ForeColor | Читает или устанавливает цвет изображения (обычно цвет текста) для Web-элемента управления |
| Height Width | Читает или устанавливает высоту и ширину Web-элемента управления |
| TabIndex | Читает или устанавливает индекс перехода по табуляции для Web-элемента управления |
| ToolTip | Читает или устанавливает значение-индикатор, указывающее необходимость отображения подсказки при задержке указателя мыши на изображении Web-элемента управления |

Скорее всего, эти свойства будут для вас понятны, так что вместо примеров их использования давайте немного сместим акценты и проверим в действии ряд элементов управления Web-формы ASP.NET.

**Категории Web-элементов управления ASP.NET**

Типы в System.Web.UI.WebControls можно разбить на несколько больших категорий.

*•*Простые элементы управления

*•*Элементы управления с расширенными возможностями

• Элементы управления для работы с источниками данных

• Элементы управления для контроля ввода

• Элементы управления для входа в систему

*Простые элементы управления*называются так потому, что они являются Web-элементами управления ASP.NET, отображающимися в стандартные HTML-элементы (кнопки, списки, гиперссылки, контейнеры изображений, таблицы и т.д.). Далее мы имеем небольшое множество так называемых *элементов управления с расширенными возможностями,*для которых нет прямого эквивалента среди HTML-элементов (это, например, Calendar, TreeView, wizard и т.д.). *Элементы управления для работы с источниками данных*являются элементами, для заполнения которых обычно требуется соединение с источником данных. Лучшим (и наиболее интересным) примером такого элемента управления ASP.NET является, наверное, GridView. Другими членами этой категории являются так называемый "ротатор" и элемент управления DataList. *Элементы управления для контроля ввода*являются серверными элементами управления, автоматически генерирующими JavaScript-код клиента для проверки вводимых в форму данных. Наконец, библиотеки базовых классов ASP.NET 2.0 предлагают целый ряд элементов управления, связанных с решением проблем безопасности, Эти элементы интерфейса инкапсулируют все особенности регистрации доступа к узлу, предлагая, в частности, сервис ввода и получения пароля и поддержку ролей пользователей.

**Замечание.**Поскольку в этой книге не предлагается обсуждение системы безопасности .NET, здесь не будут обсуждаться и соответствующие новые элементы управления. Если вам потребуется подробная информация по проблемам безопасности ASP.NET 2.0, обратитесь к книге Dominic Selly, Andrew Troelsen and Tom Barnaby, *Expert ASP.NET 2.0 Advanced Application Design*(Apress, 2006).

**Несколько слов о System.Web.UI.HtmlControls**

Вообще говоря, есть два разных набора Web-элементов управления, предлагаемых в рамках дистрибутива .NET 2.0. В дополнение к Web-элементам управления ASP.NET (из пространства имен System.Web.UI.WebControls), библиотеки базовых классов предлагают также элементы System.Web.UI.HtmlControls.

HTML-элементы управления представляют собой коллекцию типов, позволяющих использовать традиционные элементы управления HTML на странице Web-формы. Однако, в отличие от стандартных HTML-дескрипторов, HTML-элементы управления являются сущностями ООП, которые могут быть настроены для выполнения на сервере и поэтому поддерживают серверную обработку событий. В отличие от Web-элементов управления ASP.NET, HTML-элементы управления по своей природе очень просты и имеют не слишком широкие возможности, аналогичные стандартным дескрипторам HTML (HtmlButton, HtmlInputControl, HtmlTable и т.д.).

HTML-элементы управления предлагают открытый интерфейс, "имитирующий" стандартные HTML-атрибуты. Например, чтобы получить информацию из области ввода, вы должны использовать свойство Value, а не свойство Text, как в случае Web-элементов. Поскольку HTML-элементы управления обладают не такими богатыми возможностями, как Web-элементы управления ASP.NET, далее в этой книге HTML-элементы управления упоминаться не будут. Если вы захотите изучить эти типы, обратитесь к документации .NET Framework 2.0 SDK.

**Создание простого Web-узла ASP.NET 2.0**

Ограниченный объем книги не позволяет здесь описать особенности всех Web-элементов управления, входящих в доставку ASP.NET 2.0 (для этого требуется отдельная и довольно объемная книга). Но чтобы проиллюстрировать работу с paзличными Web*-*элементами управления ASP.NET, следующим нашим заданием в этой главе будет создание Web-узла, демонстрирующего использование следующих возможностей.

• Работа с шаблонами страниц

*•*Работа с элементом управления Menu

• Работа с элементом управления GridView

• Работа с элементом управления Wizard.

При работе с примером не забывайте о том, что элементы управления Web-формы инкапсулируют возможности генерирования соответствующих HTML-дескрипторов и следуют модели Windows Forms. Для начала создайте новое Web-приложение ASP.NET с названием AspNetCarSite.

**Работа с шаблоном страниц**

Вы, несомненно, знаете, что многие Web-узлы предлагают страницы, выдержанные в одном стиле (такие страницы имеют общую систему меню, общие элементы оформления верхней и нижней частей страниц, непременно содержат фирменный знак компании и т.д.). В ASP.NET 1.x разработчики широко использовали UserControl и Web-элементы управления, чтобы определить содержимое, которое должно было использоваться на многих страницах. И хотя UserControl и Web-элементы управления остаются доступными для использования в ASP.NET 2.0, теперь для решения указанных задач предлагается использовать *шаблоны страниц.*

Упрощенно говоря, шаблон страницы отличается от обычной страницы ASP.NET почти исключительно только тем, что он размещается в файле \*.master. Сами по себе шаблоны страниц не являются видимыми для браузера клиента (фактически среда выполнения ASP.NET не обслуживает эту часть Web-содержимого). Шаблоны страниц определяют общий каркас пользовательского интерфейса, совместно используемый всеми страницами (или подмножеством страниц) узла. Кроме того, страница \*.master определяет различные дескрипторы-заполнители, получающие дополнительное содержимое в файле \*.aspx. В результате получается общий, унифицированный пользовательский интерфейс.

Добавьте в свой Web-узел новый шаблон страницы (выбрав Web Site→Add New Item из меню) и рассмотрите его исходное определение.

**‹%@ Master Language="C#" AutoEventWireup="true" CodeFile="MasterPage.master.cs" Inherits="MasterPage" %›**

‹!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/ TR/xhtml11/DTD/xhtml11.dtd"›

‹html xmlns="http://www.w3.org/1999/xhtml"›

 ‹head runat="server"›

  ‹title›Untitled Page‹/title›

 ‹/head›

 ‹body›

  ‹form id="form1" runat="server"›

   ‹div›

**‹asp**:**contentplaceholder id="ContentPlaceHolder1" runat="server"›**

**‹/asp:contentplaceholder›**

   ‹/div›

  ‹/form›

 ‹/body›

‹/html›

Первым интересным элементом здесь является новая директива ‹%@Master%›. По большей части эта директива поддерживает те же атрибуты, что и ‹%@Page%›. Например, обратите внимание на то, что по умолчанию шаблон страницы предполагает использование файла внешнего кода поддержки (который, строго говоря, не обязателен). Подобно типам Page, шаблоны страниц получаются из специального базового класса, которым в данном случае является MasterPage.

public partial class MasterPage: System.Web.UI.MasterPage {

 protected void Page\_Load(object sender, EventArgs e) {

 }

}

Важно знать о том, что атрибуты, определенные директивой ‹%@Master%›, не "перетекают" в связанные файлы \*.aspx. Поэтому вы можете, например, использовать C# в рамках шаблона страниц, а для связанного файла \*.aspx использовать Visual Basic .NET.

Другим интересным элементом является ‹asp:contentplaceholder›. Эта область шаблона представляет элемент пользовательского интерфейса в связанном файле \*.aspx, а не содержимое самого шаблона страниц. Если обработать файл \*.aspx в рамках этой части шаблона, то контекст дескрипторов ‹asp:contentplaceholder› и ‹/asp:contentplaceholder› окажется пустым. Однако при желании вы можете наполнить эту область различными Web-элементами управления, которые будут функционировать в качестве элементов пользовательского интерфейса, используемых по умолчанию в том случае, когда данный файл \*.aspx узла не предложит свое конкретное содержимое. Для этого примера мы предполагаем, что все страницы \*.aspx узла предоставляют подходящее пользовательское содержимое.

**Замечание.**Страница \*.master может определять столько заместителей содержимого, сколько необходимо. Также страница \*.master может содержать дополнительные вложенные страницы \*. master.

Как и следует ожидать, в Visual Studio 2005 имеется возможность построить общий интерфейс файла \*.master с помощью тех же инструментов проектирования, что и в случае построения файлов \*.aspx. Для своего узла добавьте информирующую надпись Label (чтобы использовать ее для общего приветственного сообщения), элемент управления AdRotator (который будет случайным образом отображать одно из двух изображений) и элемент управлений Menu (чтобы позволить пользователю перейти к другим частим узла).

**Работа с элементом управления Menu**

ASP.NET 2.0 предлагает несколько новых Web-элементов управления, которые позволяют реализовать возможности навигации в пределах узла. Это SiteMapPath, TreeView и Menu. Как вы можете догадаться, эти Web-элементы могут быть настроены множеством способов. Например, каждый из этих элементов управления мо-*Ш&т*динамически генерировать, свои строки с помощью внешнего XML-файла или источник данных, Но для нашего типа Menu мы просто укажем три значения непосредственно.

В режиме проектирования Web-страницы, выберите элемент управления Menu, активизируйте встроенный редактор этого элемента (используй маркер в верхнем углу элемента) и выберите Edit Menu Items. Добавьте три корневых элемента Начало обзора, Создать машину и Ассортимент. Перед закрытием диалогового окна установите для свойства NavigateUrl каждого элемента ссылки на следующие страницы (которые еще не созданы).

• *Начало обзора:*Default.aspx

• *Создать машину:*BuildCar.aspx

• *Ассортимент:*Inventory.aspx

Этого будет достаточно, чтобы ваш элемент Menu позволял перейти к другим страницам узла. Выполнить дополнительные действия в случае выбора пользователем данного пункта меню можно с помощью обработки события MenuItemClick. Для нашего примера в этом необходимости нет, но вы должны знать, что с помощью поступающего параметра MenuEventArgs можно определить, какой пункт меню был выбран.

**Работа с AdRotator**

Роль элемента AdRotator ASP.NET заключается в случайном отображении изображений в некоторой позиции в окне браузера. Непосредственно после размещения AdRotator в окне проектирования он отображается в виде пустого заместителя элемента. Функционально этот элемент управления не сможет выполнять свою задачу до тех пор, пока вы не назначите свойству AdvertisementFile ссылку на файл, описывающий все изображения. Для нашего примера источником данных будет простой XML-файл с именем Ads.Xml.

Добавив этот новый XML-файл в узел, укажите в нем уникальный элемент ‹Ad› для каждого изображения, которое требуется отобразить. Как минимум, каждый элемент ‹Ad› должен указать изображение для отображения (ImageUrl), адрес URL для перехода при выборе данного изображения (TargetUrl), текст, появляющийся при размещении указателя мыши на изображении (AlternateText) и "вес" изображения (Impressions).

‹Advertisеments›

 <Ad>

  ‹ImageUrl›SlugBug.jpg‹/ImageUrl›

  ‹TargetUrl›http://www.Cars.com‹/TargetUrl›

  ‹AlternateText›Ваша новая машина?‹/AlternateText›

  ‹Impressions›80‹/Impressiоns›

 ‹/Ad›

 ‹Ad›

  ‹ImageUrl›car.gif‹/ImageUrl›

  ‹TargetUrl›http://www.CarSuperSite.com‹/TargetUrl›

  ‹AlternateText›Нравится эта машина?‹/AlternateText›

  ‹Impressions›80‹/Impressions›

 ‹/Ad›

‹/Advertisements›

Теперь можно связать XML-файл с элементом управления AdRotator с помощью свойства AdvertisementFile (в окне свойств).

‹asp:AdRotator ID="myAdRotator" runat="server" AdvertisementFile="~/Ads.xml"/›

Позже, когда вы запустите это приложение и направите вторичный запрос странице, вам будет показано одно из двух изображений, выбранное случайно. На рис. 23.21 показан исходный вид шаблона страницы.

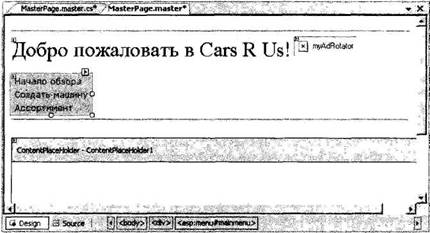


Рис. 23.21. Шаблон страницы

**Определение страницы Default.aspx**

После установки шаблона страниц вы можете приступить к созданию индивидуальных страниц \*.aspx, определяющих в совокупности с дескриптором ‹asp: contentplaceholder› шаблона содержимое пользовательского интерфейса. При создании нового Web-узла среда разработки Visual Studio 2005 автоматически создает начальный файл \*.aspx, но в исходном своем состоянии этот файл не может быть соединен с шаблоном страниц.

Причина в том, что именно файл \*.master определяет раздел ‹form› результирующей HTML-страницы. Поэтому существующую область ‹form› в файле \*. aspx нужно заменить на ‹asp:content›. Переключитесь в режим Source дли Default.aspx и замените имеющуюся разметку следующей.

‹%@ Page Language="C#" **MasterPageFile="~/MasterPage.master"**AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="\_Default" Title="Untitled Page" %›

‹asp:Content ID="Content1" **ContentPlaceHolderID="ContentPlaceHolder1"** Runat="Server"›

‹/asp:Content›

Прежде всего обратите внимание на то, что директива ‹%Page%› здесь получает новый атрибут MasterPageFile, которому присвоено значение вашего файла \*.master. Также отметьте, что значение ContentPlaceHolderID идентично значению элемента ‹asp:contentplaceholder› из файла шаблона.

После переключения обратно в режим проектирования (режим Design) вы обнаружите, что пользовательский интерфейс шаблона страницы стал видимым. Область содержимого также видима, хотя в данный момент она пуста. У нас нет необходимости строить сложный интерфейс для области содержимого Default.aspx. поэтому для примера просто добавьте несколько типов Label, чтобы отобразить основные инструкции узла (рис. 23.22).

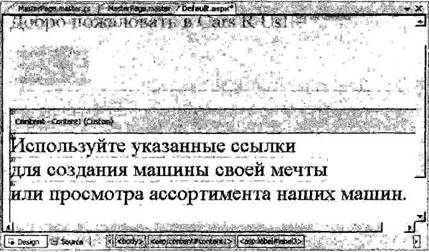


Рис. 23.22. Страница содержимого Default.aspx

Если теперь запустить проект на выполнение то вы увидите, что содержимое интерфейсов файлов \*.master и Default.aspx сливается в едином HTML-потоке. Как видно из рис. 23.23, конечный пользователь не увидит даже признаков того, что существует шаблон страницы.

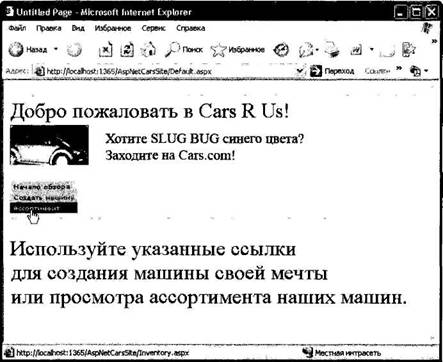


Рис. 23.23. Страница Cars R Us, открываемая по умолчанию

**Создание страницы Inventory**

Чтобы добавить в проект страницу содержимого Inventory.aspx, откройте в среде разработки страницу \*.master и выберите WebSite→Add Content Page из меню (если файл \*.master не является активным элементом, этот пункт меню не предлагается). Роль страницы Inventory заключается в отображении содержимого таблицы Inventory базы данных Cars в рамках элемента управления GridView.

Вы, наверное, знаете, что элемент управления GridView не ограничивается представлением данных только для чтения. Этот элемент можно настроить так, чтобы поддерживались сортировка, перелистывание и редактирование данных. При желании вы имеете возможность выполнить серверную обработку серии событий, добавив соответствующий программный код ADO.NET. Этот элемент ASP.NET 2.0 заменяет соответствующий элемент ASP.NET 1.x c его "менталитетом нулевого программного кода.

С помощью нескольких щелчков кнопки мыши вы можете настроить GridView на автоматический выбор, обновление и удаление записей соответствующего хранилища данных. "Нулевой" программный код сильно уменьшает объем вводимого шаблонного программного кода, но важно понимать, что эта простота сопровождается потерей контроля и не может быть рекомендована для использования в приложениях производственного уровня.

Тем не менее, для иллюстрации возможностей использования GridView в этой декларативной форме добавьте новую страницу (Inventory.aspx) и создайте в области содержимого информирующий тип Label и тип GridView. Используя встроенный редактор, выберите New Data Source из раскрывающегося списка Choose Data Source. Будет запущен мастер, который за несколько шагов поможет вам соединить компонент с соответствующим источником данных. Вот шаги, которые нужно сделать для нашего примера.

1. Выберите пиктограмму Database и укажите CarsDataSource для идентификатора источника данных.

2. Выберите базу данных Cars (если потребуется, создайте для этого новое соединение).

3. Если хотите, сохраните данные строки соединений в файле Web.config. Напоминаем (см. главу 22), что ADO.NET теперь поддерживает элемент ‹connectionStrings›.

4. Укажите SQL-оператор Select для выбора всех записей из таблицы Inventory (pиc. 23.24).

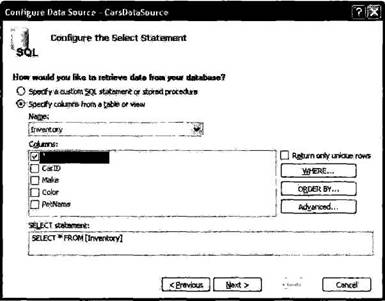


Рис. 23.24. Выбор таблицы Inventory

Если теперь проверить содержимое открывающего дескриптора элемента управления GridView, вы увидите, что свойство DataSourceID получило тот тип SqlDataSource, который вы только что определили.

‹asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False" CellPadding="4" DataKeyNames="CarID" **DataSourceID="CarsDataSource"** ForeColor="#33333З" GridLines="None"›

…

‹/asp:GridView›

Тип SqlDataSource (новый в .NET 2.0) - это компонент, инкапсулирующий информацию о хранилище данных. С учетом знаний, освоенных вами в главе 22, следующие атрибуты должны быть для вас понятными:

‹asp:SqlDataSource ID="CarsDataSource" runat="server" ConnectionString="Data Source=localhost;Initial Catalog=Cars;Integrated Security=True" ProviderName=''System.Data.SqlClient" SelectCommand="SELECT \* FROM [Inventory]"›

‹/asp:SqlDataSource›

Теперь вы можете запустить свою Web-программу, выполнить щелчок на пункте меню Ассортимент и просмотреть соответствующие данные (рис. 23.25).

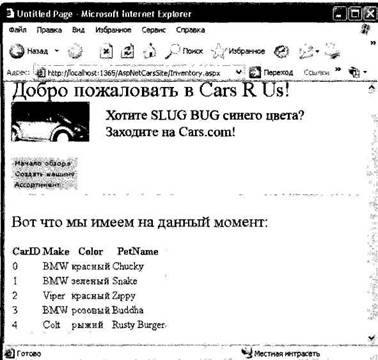


Рис. 23.25. Элемент управления GridView без дополнительного программного кода

**Разрешение сортировки и перелистывания**

Элемент управления GridView можно легко настроить для сортировки (по ссылке на имя столбца] и перелистывания страниц (по номеру или гиперссылке на следующую/предыдущую страницу). Для этого нужно активизировать встроенный редактор элемента и отметить соответствующие пункты меню (рис. 23.26).

Когда вы откроете страницу снова, вы сможете отсортировать отображаемые данные с помощью щелчка на имени соответствующего столбца и листать данные с помощью ссылок на номера страниц (конечно, при условии, что для этого достаточно записей в таблице Inventory).

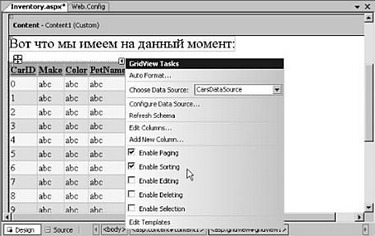


Рис. 23.26. Разрешение перелистывания и сортировки страниц

**Разрешение редактирования на месте**

Заключительным штрихом оформления этой страницы будет поддержка элементом управления GridView редактирования данных. Для этого откройте встроенный редактор для SqlDataSource и выберите Configure Data Source. Пропустите первые два окна этого мастера, а на шаге 3 щелкните на кнопке Advanced и отметьте первый вариант настройки (рис. 23.27).

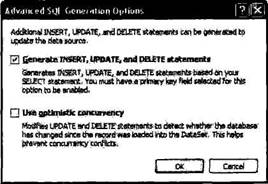


Рис. 23.27. Автоматическое генерирование SQL-операторов

Если теперь рассмотреть HTML-определение элемента управления, то вы увидите, что для SqlDataSource определены DeleteCommand, InsertCommand и UpdateCommand (с помощью параметризованных запросов).

‹asp:SqlDataSource ID="CarsDataSource" runat="server" ConnectionString="Data Source=localhost;Initial Catalog=Cars;Integrated Security=True"  ProviderName="System.Data.SqlClient"**SelectCommand**="SELECT \* FROM [Inventory]" **DeleteCommand**="DELETE FROM [Inventory] WHERE [CarID] = @original\_CarID" **InsertCommand**="INSERT INTO [inventory] ([CarID], [Make], [Color], [PetName]) VALUES (@CarID, @Make, @Color, @PetName)" **UpdateCommand**="UPDATE [Inventory] SET [Make] = @Make, [Color] = @Color, [PetName]=@PetName WHERE [CarID]=@original\_CarID"›

…

‹/asp:SqlDataSource›

Также будет присутствовать компонент SqlDataSource, который обеспечит дополнительную разметку, определяющую объекты параметров для параметризованных запросов.

‹DeleteParameters›

 ‹asp:Parameter Name="original\_CarID" Type="Int32" /›

‹/DeleteParameters›

‹UpdateParameters›

 ‹asp:Parameter Name="Make" Type="String" /›

 ‹asp:Parameter Name="Color" Type="String" /›

 ‹asp:Parameter Name="PetName" Type="String" /›

 ‹asp:Parameter Name="original\_CarID" Type="Int32" /›

‹/UpdateParameters›

‹InsertParameters›

 ‹asp:Parameter Name="CarID" Type="Int32" /›

 ‹asp:Parameter Name="Make" Type="String" /›

 ‹asp:Parameter Name="Color" Type="String" /›

 ‹asp:Parameter Name="PetName" Type="String" /›

‹/InsertParameters›

Заключительным шагом является разрешение поддержки редактирования и удаления данных с помощью встроенного редактора GridView (рис. 23.28).

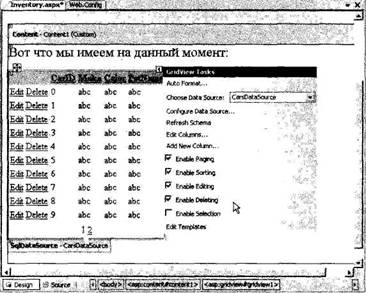


Рис. 23.28. Разрешение поддержки редактирования и удаления данных

Достаточно очевидно, что при новом обращении к странице Inventorу.aspx вы сможете редактировать и удалять записи (рис. 23.29).

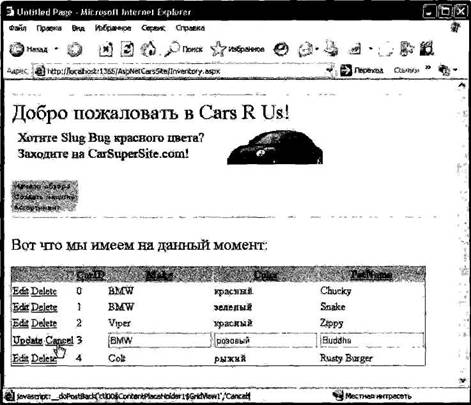


Рис. 23.29. Таблица всех таблиц

**Создание страницы BuildCar**

Последним нашим заданием в этом примере будет создание страницы BuildCar.aspx. Добавьте ее в свой проект (выбрав Web Site→Add Content Page из меню). Эта страница будет содержать Web-элемент управления Wizard ASP.NET 2.0, который обеспечит простой способ прохождения конечного пользователя через серию связанных шагов. Здесь соответствующие шаги будут имитировать выбор покупателем автомобили с нужными ему характеристиками.

Поместите в область содержимого информирующую надпись и элемент управления Wizard. Затем активизируйте встроенный редактор для Wizard и щелкните на ссылке Add/Remove WizardSteps (Добавить или удалить шаги мастера). Добавьте четыре шага, как показано на рис. 23.30.

После определения этих шагов вы увидите, что Wizard определяет области с пустым содержимым, куда вы можете перетащить элементы управления, необходимые для выбранного в настоящий момент шага.



Рис. 23.30. Конфигурация шагов Wizard

В нашем случае добавьте для шагов следующие элементы пользовательского интерфейса (не забудьте указать соответствующее значение ID для каждого элемента, используя окно свойств).

• *Выберите модель:*элемент управления TextBox

• *Выберите цвет:*элемент управления ListBox

• *Укажите название:*элемент управления TextBox

• *Укажите дату доставки:*элемент управления Calendar

Элемент управления ListBox является единственным элементом интерфейса Wizard, требующим дополнительной настройки. Выберите этот элемент в окне проектирования (убедившись, что перед этим вы выбрали ссылку *Выберите цвет)*и укажите для элемента набор цветов с помощью свойства Items в окне свойств. После этого в контексте определения Wizard вы обнаружите разметку, похожую на следующую.

‹asp:ListBox ID="ListBoxColors" runat="server" Width="237px"›

 ‹asp:ListItem›пурпурный‹/asp:ListItem›

 ‹asp:ListItem›зеленый‹/asp:ListItem›

 ‹asp:ListItem›красный‹/asp:ListItem›

 ‹asp:ListItem›желтый‹/asp:ListItem›

 ‹asp:ListItem›светло-зеленый‹/asp:ListItem›

 ‹asp:ListItem›черный‹/asp:ListItem›

 ‹asp:ListItem›лимонный‹/asp:ListItem›

‹/asp:ListBox›

После определения каждого из шагов вы можете обработать событие FinishButtonClick для автоматически генерируемой кнопки Finish (Готово). В серверном обработчике события получите параметры выделения от каждого элемента интерфейса и постройте строку описания, которая будет назначена свойству Text дополнительного типа Label с именем lblOrder.

protected void carWizard\_FinishButtonClick(object sender, WizardNavigationEventArgs e) {

**// Получение значений.**

 string order = string.Format("{0}, ваш {1} {2}, будет доставлен {3}.",

  txt.CarPetName.Text,

  ListBoxColors.SelectedValue,

  txtCarModel.Text,

  carCaLendar.SelectedDate.ToShortDateString());

**// Присваивание значения надписи.**

 lblOrder.Text = order;

}

Итак, ваш узел AspNetCarSite готов. На рис. 23.31 показан элемент Wizard в действии.

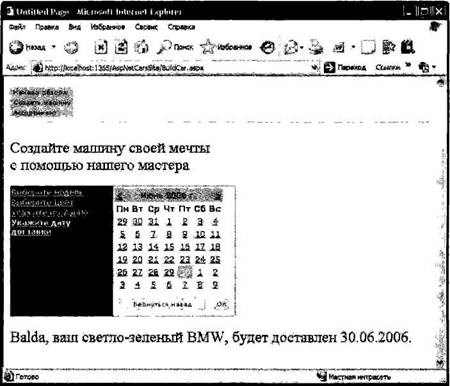


Рис. 23.31. Wizard в действии

На этом завершается наш обзор Web-элементов управления. Не сомневайтесь. что имеется очень много других элементов, которые здесь охвачены не были, Однако теперь вы должны чувствовать себя довольно уверенно при использовании основных приемов данной модели программирования. А в завершение этой главы мы рассмотрим элементы управления, связанные с контролем вводимых данных.

**Исходный код**. Файлы AspNetCarsSite размещены в подкаталоге, соответствующем главе 23.

**Роль элементов управления, связанных с контролем ввода**

Заключительной группой рассматриваемых здесь элементов управления Web-формы являются так называемые *элементы контроля ввода.*В отличие от остальных рассмотренных нами элементов управления Web-формы, элементы контроля ввода используются для генерирования не HTML-кода, а JavaScript-кода клиента (и, возможно, программного кода сервера), предназначенного для проверки правильности вводимых в форму данных. Как показано в начале этой главы, контроль ввода на стороне клиента полезен тем, что в этом случае вы можете обеспечить выполнение различных ограничений для вводимых данных на месте, перед тем как возвратить данные Web-серверу, в результате чего число ресурсоемких обращений к серверу уменьшается. В табл. 23.11 предлагаются описания элементов управления ASP.NET, связанных с контролем ввода.

Таблица 23.11. Элементы контроля ввода ASP.NET

|  |  |
| --- | --- |
| **Элемент управления** | **Описание** |
| CompareValidator | Выполняет проверку значения одного элемента управления на равенство фиксированной константе или значению другого элемента управления |
| CustomValidator | Позволяет построить функцию пользовательского контроля ввода для данного элемента управления |
| RangeValidator | Проверяет принадлежность значения заданному диапазону значений |
| RegularExpressionValidator | Проверяет соответствие значения соответствующего элемента управления заданному шаблону регулярного выражения |
| RequiredFieldValidator | Гарантирует, что данный элемент управления не останется пустым (т.е. будет содержать значение) |
| ValidationSummary | Отображает резюме всех ошибок проверки ввода на странице в формате списка, списка с буллитами или формате единого абзаца. Ошибки могут отображаться "на месте" и/или во всплывающем окне сообщения |

Все элементы контроля ввода, в конечном счете, получаются из общего базового класса System.Web.UI.WebControls.BaseValidator, поэтому они должны иметь множество общих свойств. Описания ряда таких свойств предлагаются в табл. 23.12.

Чтобы продемонстрировать основы работы с элементами контроля ввода, создайте новый Web-узел с именем ValidatorCtrls. Сначала поместите на страницу четыре типа TextBox (с четырьмя соответствующими информирующими типами Label). Затем по соседству с каждым полем разместите типы RequiredFieldValidator, RangeValidator, RegularExpressionValidator и CompareValidator. Наконец, добавьте одну кнопку (Button) и надпись (Label), рис. 23.32.

**Таблица 23.12.**Общие свойства элементов контроля ввода ASP.NET

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| СontrolToValidiate | Читает или устанавливает имя элемента управления, который необходимо контролировать |
| Display | Читает или устанавливает значение, характеризующее вид отображения сообщения об ошибке для элемента контроля ввода |
| EnableClientScript | Читает или устанавливает признак активизации контроля ввода на стороне клиента |
| ErrorMessage | Читает или устанавливает текст сообщения об ошибке |
| ForeColor | Читает или устанавливает цвет сообщения, отображаемого при отрицательном исходе проверки ввода |

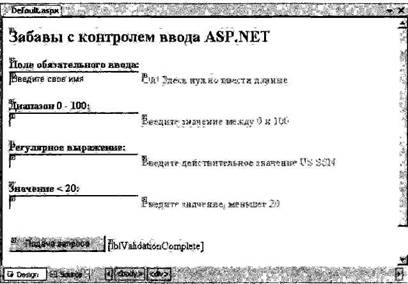


Рис. 23.32. Элементы, которые придется контролировать

Теперь у вас есть пользовательский интерфейс, и мы с вами можем заняться настройкой каждого из его элементов.

**Элемент RequiredFieldValidator**

Настроить RequiredFieldValidator очень просто. В окне свойств Visual Studio 2005 установите для свойств ErrorMessage и ControlToValidate нужные значения. Определение \*.aspx должно быть таким.

‹asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server" **ControlToValidate="txtRequiredField" ErrorMessage="Ой! Здесь нужно ввести данные.**"›

‹/asp:RequiredFieldValidator*›*

Одной приятной особенностью RequiredFieldValidator является то, что этот элемент поддерживает свойство InitialValue. Это свойство используется для того, чтобы гарантировать ввод пользователем любого значения, отличного от начального значения соответствующего поля TextBox. Например, если пользователь обращается к странице впервые, вы можете настроить TextBox на отображение значения "Введите свое имя". Если не установить свойство InitialValue элемента RequiredFieldValidator, среда выполнения будет предполагать, что строка "Введите свое имя" является действительной, Для гарантии того, что данный TextBox пройдет контроль только тогда, когда пользователь введет строку, отличную от "Введите свое имя", вы должны настроить элемент так, как показано ниже.

‹asp:RequiredFieldValidator ID="RequiredFieldValidator1" гunat="server" ControlToValidate="txtRequiredField" ErrorMessage="Ой! Здесь нужно ввести данные." **InitialValue="Введите свое имя"›**

‹/asp:RequiredFieldValidator›

**Элемент RegularExpressionValidator**

Элемент RegularExpressionValidator может использоваться тогда, когда требуется сравнение введенных символов с некоторым шаблоном. Так, для гарантии того, что поле данного TextBox содержит действительный номер социальной страховки США (US SSN), можно определить элемент так, как предлагается ниже.

‹asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server" ControlToValidate="txtRegExp" ErrorMessage="Введите действительное значение US SSN."**ValidationExpression="\d{3}-\d{2}-\d{4}**"›

‹/asp:RegularExpressionValidator›

Здесь обратите внимание на то, что RegularExpressionValidator определяет свойство ValidationExpression. Если вы никогда не работали с регулярными выражениями, вам для понимания этого примера нужно знать только то, что регулярные выражения используются для проверки соответствия заданному строковому шаблону. Указанное здесь выражение "\d{3}-\d{2}-\d{4}" соответствует стандартному номеру социальной страховки в США, имеющему вид *ххх-хх-хххх*(где x означает любую цифру).

Это конкретное регулярное выражение вполне очевидно, но предположим, что нам нужно проверить соответствие действительному телефонному номеру Японии. Выражение, необходимое в данном случае, является более сложным – оно имеет вид "(0\d{1,4}-|\(0\d{1,4}\)?)?\d{1,4}-\d{4}". Поэтому очень хорошо, что при выборе свойства ValidationExpression в окне свойств вы имеете возможность выбрать подходящее значение из встроенного набора часто используемых регулярных выражений (рис. 23.33).

**Замечание.**Если вам действительно нужны регулярные выражения, то знайте, что платформа .NET для программной работы с регулярными выражениями предлагает два пространства имен (System.Text.RegularExpressions и Systern.Web.RegularExpressions).

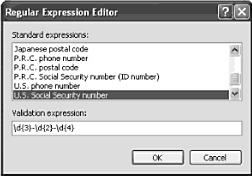


Рис. 23.33. Создание регулярного выражения с помощью Visual Studio 2005

**Элемент RangeValidator**

В дополнение к Свойствам MinimumValue и MaximumValue, элементы RangeValidator имеют свойство Type. Чтобы выполнить проверку ввода пользователя на соответствие заданному диапазону целых чисел, для этого свойства нужно указать Integer (что не является значением по умолчанию!).

‹asp:RangeValidator ID="RangeValidator1" runat="server" ControlToValidate="txtRange" ErrorMessage="Введите значение между 0 и 100." **MaximumValue="100" MinimumValue="0" Type="Integer**"›

‹/asp:RangeValidator›

Элемент RangeValidator можно использовать и тогда, когда нужно проверить, что введенное значение принадлежит заданному диапазону денежных значений, значений дат, чисел с плавающей точкой или строковых данных (это и есть значение, устанавливаемое по умолчанию).

**Элемент CompareValidator**

Наконец, обратим ваше внимание на то, что CompareValidator поддерживает свойство Operator.

‹asp:CompareValidator ID="CompareValidator1" runat="server" ControlToValidate="txtComparison" ErrorMessage="Введите значение, меньшее 20." **Operator="LessThan" ValueToCompare="20"**›

*‹/*asp:CompareValidaror*›*

Поскольку задачей этого элемента контроля ввода является сравнение значения в текстовом блоке с другим значением при помощи бинарного оператора, не удивительно то, что свойству Operator можно назначить такие значения, как LessThan (меньше), GreaterThan (больше), Equal (равно) и NotEqual (не равно). Также заметьте, что для указания значения, с которым производится сравнение, используется ValueToCompare.

**Замечание.**С помощью свойства ControlToValidate элемент CompareValidator можно настроить на сравнение со значением из другого элемента управления Web-формы (а не с конкретным "жестко" заданным значением).

Чтобы завершить создание программного кода для страницы, обработайте событие Click для типа Button и проинформируйте пользователя о завершении работы программы контроля.

protected void btnPostback\_Cliсk(object sender, EventArgs e) {

 lblValidationComplete.Text = "Вы прошли контрольную проверку!";

}

Теперь откройте созданную страницу с помощью любого браузера. Никаких заметных изменений вы не увидите. Но если вы попытаетесь щелкнуть на кнопке Подача запроса после ввода неподходящих данных, появится ваше сообщение об ошибке. После ввода корректных данных сообщения об ошибке исчезнут, и соответствующий запрос будет отправлен.

Если взглянуть на HTML-код, отображаемый браузером, вы увидите, что элементы контроля ввода генерируют JavaScript-функцию клиента, использующую специальную библиотеку JavaScript-функций (она содержится в файле WebUIValidation.js), которая автоматически загружается на машине пользователя. По завершении проверки ввода данные формы направляются серверу, где среда выполнения ASP.NET снова выполнит *те же*проверки уже на Web-сервере (для гарантии того, что не произошло никаких искажений при передаче данных).

В связи со всем сказанным, если HTTP-запрос был послан браузером, не поддерживающим JavaScript клиента, весь контроль будет происходить на сервере. Поэтому вы можете программировать элементы контроля ввода без учета возможностей целевого браузера – возвращенная HTML-страница направит задачу контроля ошибок обратно Web-серверу.

**Создание отчетов по проверкам**

Заключительной из рассматриваемых здесь задач контроля ввода будет использование элемента ValidationSummary. В настоящий момент каждый из ваших элементов контроля отображает сообщение об ошибке в том месте, где это было определено на этапе разработки. Во многих случаях это будет как раз то, что вам требуется. Однако в сложных формах с многочисленными элементами ввода вы не захотите видеть неожиданно всплывающие то тут, то там яркие сообщения. Используя тип ValidationSummary, вы можете заставить все типы контроля ввода отображать сообщения об ошибках в строго отведенном для этого месте на странице.

Первым шагом при этом является добавление ValidationSummary в файл \*.aspx. Дополнительно можно установить свойство HeaderText этого типа, а также значение DisplayMode, которое по умолчанию задает представление сообщений об ошибках в виде списка с буллитами.

‹asp:ValidationSummary id="ValidationSummary1" style="Z-INDEX: 123; LEFT: 152px; POSITION: absolute; TOP: 320px" runat="server" Width="353px" **HeaderText=**"Элементы ввода, требующие корректировки."›

‹/asp:ValidationSummary›

Затем для каждого элемента контроля ввода на странице (для RequiredFieldValidator, RangeValidator и т.д.) нужно установить свойство Display равным значению None. Это гарантирует, что вы не увидите дубликатов сообщений об ошибках для одного и того же случая неудачного завершения проверки (когда одно сообщение отображается в резюмирующем списке, а другое – в месте размещения элемента контроля ввода).

Наконец, если требуется отобразить сообщения об ошибке с помощью MessageBox клиента, то установите для свойства ShowMessageBox значение true (истина), а для свойства ShowSummary – значение false (ложь).

**Исходный код**. Проект ValidatorCtrls размещен в подкаталоге, соответствующем главе 23.

**Резюме**

Создание Web-приложений требует иного подхода по сравнению с тем, который иcпользуется для создания "традиционных" приложений. В начале этой главы был предложен краткий обзор фундаментальных составляющих Web-разработки, к которым можно отнести HTML, HTTP, сценарии клиента и сценарии сервера при использовании классической технологии ASP.

Значительная часть главы была посвящена рассмотрению архитектуры страницы ASP.NET. Вы увидели, что с каждым файлом \*.aspx в проекте связан некоторый класс, производный от System.Web.UI.Page. С помощью такого подхода ASP.NET позволяет строить более пригодные для многократного использования системы, соответствующие принципам ООП. В этой главе рассматривалась также использование шаблонов страниц и различных Web-элементов управления (включая новые типы GridView и Wizard). Вы могли убедиться в том, что эти элементы графического интерфейса отвечают за создание подходящих дескрипторов HTML-кода, направляемого клиенту. Элементы контроля ввода являются серверными элементами, на которые возлагается задача подготовки JavaScript-кода клиента для выполнения проверки допустимости введенных в форму данных, чтобы уменьшить количество необходимых обращений к серверу.

**ГЛАВА 24. Web-приложения ASP.NET 2.0**

Предыдущая глава была посвящена композиции страниц ASP.NET и поведению содержащихся в них Web-элементов управления. На основе полученных знаний в этой главе мы рассмотрим роль типа HttpApplication. Вы увидите, что функциональные возможности HttpApplication позволяют выполнять перехват ряда событий, что дает возможность рассматривать Web-приложения, скорее, как связную единицу, а не как набор автономных файлов \*.aspx.

В дополнение к исследованию типа HttpApplication в этой главе также обсуждается важная тема управления состоянием объектов. Здесь вы узнаете о роли данных состояния представлений и элементов управления, о переменных уровня сеанса и приложения, а также о связанной с состояниями конструкции ASP.NET, которая называется *кэш приложения.*После основательного изучения предлагаемых платформой .NET приемов управления состояниями, в конце главы мы обсудим роль файла Web.config и различные приемы изменения конфигурации приложений.

**Проблема состояния**

В начале предыдущей главы было указано, что HTTP является сетевым протоколом, не обеспечивающим сохранение состояний. Именно этот факт делает процесс разработки Web-приложений столь отличающимся от процесса построения выполняемого компоновочного блока. Например, при создании приложения Windows Forms вы можете быть уверены в том, что любые члены-переменные, определенные в классе формы, будут существовать в памяти до тех пор, пока пользователь не прекратит работу выполняемого файла.

public partial class MainWindow: Form {

**// Данные состояния.**

 private string userFavoriteCar;

 …

}

Однако в World Wide Web вы не можете делать такие роскошные предположения. Чтобы не быть голословными, давайте создадим новый Web-узел ASP.NET (с именем SimpleStateExample), содержащий один файл \*.aspx. В файле с внешним кодом поддержки определим строковую переменную уровня страницы с именем userFavoriteCar.

public partial class \_Default: Page {

**// Данные состояния?**

 private string userFavoriteCar;

}

Далее, построим Web-интерфейс пользователя, показанный на рис. 24.1.

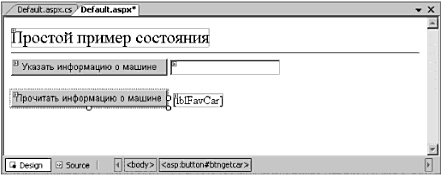


Рис. 24.1. Пользовательский интерфейс для страницы примера состояния

Обработчик события Click сервера для кнопки Указать… позволит назначить строковую переменную в соответствии со значением TextBox:

protected void btnSetCar\_Click(object sender, EventArgs e) {

**// Сохранение информации о машине.**

 userFavoriteCar = txtFavCar.Text;

}

а обработчик события Click для кнопки Прочитать… будет отображать текущее значение члена-переменной в поле элемента Label страницы.

protected void btnGetCar\_Click(object sender, EventArgs e)

**// Присваивание тексту надписи значения члена-переменной.**

 lblFavCar.Text = userFavoriteCar;

}

При построении приложения Windows Forms можно предполагать, что после установки пользователем начального значения это значение будет сохраняться в течение всего времени работы приложения. Запустив наше Web-приложение, вы, к сожалению, обнаружите, что при каждом вторичном обращении к Web-серверу значение строковой переменной userFavoriteCar снова возвращается в своему пустому начальному значению, так что поле текста Label будет постоянно оставаться пустым.

Снова подчеркнем, что HTTP не имеет никаких внутренних средств автоматического запоминания данных уже отправленного HTTP-ответа, и именно поэтому объект Page немедленно уничтожается. Когда клиент повторно обращается к файлу \*.aspx, создается новый объект Page, переустанавливающий все члены-переменные уровня страницы. Это, очевидно, и оказывается главной проблемой. Представьте себе, каким бы неудобным был процесс заказа товаров, если бы каждый раз при обращении к Web-серверу вся введенная вами информация (например, о том, что вы хотите купить) пропадала. Если вам необходимо помнить информацию о пользователях, которые регистрируются на вашем узле, вам придется использовать различные приемы сохранения состояния объектов.

**Замечание**. Эта проблема касается не только ASP.NET. Сервлеты Java, CGI-, "классические" ASP-и РНР-приложения – всем этим технологиям также приходится решать проблемы управления состоянием.

Чтобы сохранить значение строкового типа userFavoriteCar в промежутке между повторными обращениями к серверу, можно запомнить значения этого типа в сеансовой переменной. Соответствующие подробности обработки состояния сеанса будут рассматриваться в следующих разделах. Но здесь для полноты мы приводим соответствующий программный код, необходимый для текущей страницы (заметьте, что здесь больше не используется приватный член-переменная строкового типа, так что не забудьте закомментировать или просто удалить его определение).

protected void btnSetCar\_Click(object sender, EventArgs e) {

 Session["UserFavCar"] = txtFavcar.Text;

}

protected void btnGetCar\_Click(object sender, EventArgs e) {

 lblFavCar.Text = (string)Session["UserFavCar"];

}

Если выполнить приложение теперь, то информация о любимой машине в промежутке между обращениями к серверу будет сохраняться благодаря объекту HttpSessionState, обрабатываемому с помощью унаследованного свойства

**Исходный код.**Файлы примера SimpleStateExample размещены в подкаталоге, соответствующем главе 24.

**Технологии управления состоянием ASP.NET**

ASP.NET предлагает целый ряд механизмов, которые можно использовать для поддержки информации состояния в Web-приложениях. В частности, у вас на выбор есть следующие варианты.

• Использование данных состояния представлений ASP.NET.

• Использование данных состояния элементов управления ASP.NET.

• Определение переменных уровня приложения.

• Использование объектов кэширования.

• Определение переменных сеансового уровня.

• Взаимодействие с данными cookie.

Мы рассмотрим детали каждого из указанных подходов по очереди, начиная с темы состояния представлений ASP.NET.

**Роль состояния представлений ASP.NET**

Термин *состояние представлений*уже упоминался множество раз здесь и в предыдущей главе без формального определения, так что позвольте демистифицировать этот термин раз и навсегда. В рамках классической технологии ASP требовалось, чтобы Web-разработчики вручную указывали значения элементов формы в процессе построения исходящего HTTP-ответа, Например, если поступивший HTTP-запрос содержал пять текстовых блоков с конкретными значениями, файл \*.asp должен был извлечь текущие значения (с помощью коллекций Form или QueryString объекта Request) и вручную поместить их в поток HTTP-ответа (ясно, что это было весьма неудобное решение). Если разработчик этого не делал, то пользователь видел пять пустых текстовых блоков.

В ASP.NET больше не требуется вручную извлекать и указывать значения, содержащиеся в HTML-элементах, поскольку среда выполнения ASP.NET автоматически создает в форме скрытое поле (\_\_VIEWSTATE), которое включается в поток обмена между браузером и соответствующей страницей. Данные, присваиваемые этому полю, представляют собой строку в кодировке Base64, состоящую из пар имен и значений, которые характеризуют каждый элемент графического интерфейса на данной странице.

Обработчик события Init базового класса System.Web.UI.Page отвечает за чтение значений, обнаруженных в поле \_\_VIEWSTATE, и заполнение соответствующих членов-переменных в производном классе (именно поэтому по крайней мере рискованно использовать доступ к параметрам состояния Web-элемента в контексте обработчика события Init страницы).

Точно так же перед исходящим ответом браузеру данные \_\_VIEWSTATE используется для заполнения элементов формы, чтобы текущие значения HTML-элементов оставались такими же, какими они были до повторного обращения к серверу.

Очевидно, лучшим аспектом поведения ASP.NET в этом отношении является то, что все это происходит без вашего вмешательства. При этом, при желании, вы можете изменить или отключить такое принятое по умолчанию поведение. Чтобы понять, как это сделать, давайте рассмотрим конкретный пример использования данных состояния представлений.

**Демонстрация использования состояния представлений**

Создайте новое Web-приложение ASP.NET с названием ViewStateApp. В исходную страницу \*.aspx добавьте один Web-элемент управления ASP.NET ListBox и один тип Button. Обработайте событие Click для Button, чтобы обеспечить пользователю возможность вторичного обращения к Web-серверу.

protected void btnDoPostBack\_Click(object sender, EventArgs e) {

**// Для вторичного обращения к серверу.**

}

Теперь, используя окно свойств Visual Studio 2005, получите доступ к свойству Items и добавьте в ListBox четыре элемента ListItem. Результат должен быть примерно таким.

‹asp:ListBox ID="myListBox" runat="server"›

 ‹asp:ListItem›Элемент 1‹/asp:ListItem›

 ‹asp:ListItem›Элемент 2‹/asp:ListItem›

 ‹asp:ListItem›Элемент 3‹/asp:ListItem›

 ‹asp:ListItem›Элемент 4‹/asp:ListItem›

‹/asp:ListBox›

Заметьте, что здесь элементы ListBox в файле \*.aspx указаны явно. Вы уже знаете, что все элементы ‹asp:›, обнаруженные в пределах HTML-формы, автоматически обновляют свое HTML-представление перед отправкой HTTP-ответа (конечно, если они при этом имеют атрибут runat="server").

Директива ‹%@Page%› имеет необязательный атрибут enableViewState, который по умолчанию установлен равным true. Чтобы отменять такое поведение, просто измените содержимое директивы ‹\*@Page%› так, как предлагается ниже.

‹%@Page **EnableViewState ="false"**Language="C#" AutoEventWireup="true" CodeFilе="Dеfault.aspx.cs" Inherits="\_Default" %›

Но что же в действительности означает отключение учета состояния представлений? Ответ на этот вопрос зависит от многих факторов. В соответствии с предложенным выше определением данного термина вы могли бы предположить, что при отключении учета состояния представлений для файла \*.aspx значения ListBox не будут сохраняться при вторичных запросах к Web-серверу. Однако, выполнив приложение в том виде, в каком это приложение находится сейчас, вы можете с удивлением обнаружить, что информация в ListBox сохраняется независимо от того. сколько раз вы повторно обращаетесь к странице. Фактически, если рассмотреть исходный HTML-код, возвращаемый браузеру, вы увидите, что скрытое поле \_\_VIEWSTATE там *все равно*присутствует.

‹input type="hidden" name="\_\_VIEWSTATE" id="\_\_VIEWSTATE" value= **"/wEPDwUKMTYlMjcхNTсxNmRkОXbNzW5+R2VDhNWtEtHMM+yhxvU=" /›**

Причиной, по которой строка состояния представлений не исчезает, является то, что файл \*.aspx явно определяет элементы ListBox в контексте HTML-дескриптора *‹*form›. Поэтому элементы ListBox будут автоматически генерироваться при каждом ответе Web-сервера клиенту.

Но предположим, что значения вашего типа ListBox динамически указываются в файле внешнего кода поддержки, а не в рамках HTML-определения ‹form›. Сначала удалите декларации ‹asp:ListItem› из имеющегося файла \*.aspx.

‹asp:ListBox ID="myListBox" runat="server"›

‹/asp:ListBox›

Затем задайте элементы списка в обработчике события Load в рамках файла с внешним кодом поддержки.

protected void Page\_Load(object sender, EventArgs e) {

 if (!IsPostBack) {

**// Динамическое заполнение ListBox.**

  myListBox.Items.Add("Элемент 1");

  myListBox.Items.Add("Элемент 2");

  myListBox.Items.Add("Элемент 3");

  myListBox.Items.Add("Элемент 4");

 }

}

Для обновленной страницы вы обнаружите, что при первом запросе страницы браузером значения в ListBox присутствуют и учитываются. Однако после вторичного запроса ListBox становится *пустым.*Первым правилом состояния представлений ASP.NET является то, что они учитывается только при наличии элементов, чьи значения динамически генерируются программным кодом. Если вы явно укажете значения в рамках дескриптора ‹form› в файле \*.aspx, состояние этих элементов всегда будет восстанавливаться при вторичных запросах (даже если вы установите для enableViewState значение false для данной страницы).

Данные состояния представлений оказываются наиболее полезными при работе с динамическими Web-элементами управления, которые должны обновляться при каждом вторичном обращении к серверу (примером такого Web-элемента является элемент управления GridView ASP.NET, заполняемый при обращении к базе данных). Если вы не отключите учет состояния представлений для страниц, содержащих такие элементы, состояние таблиц будет представлено в скрытом поле\_\_ VIEWSTATE. Сложные страницы могут содержать очень много Web-элементов управления ASP.NET, поэтому вы можете себе представить, насколько большой может быть соответствующая строка. При этом содержимое HTTP-цикла "запрос-ответ" может оказаться достаточно объемным, что может стать определенной проблемой для коммутируемых Web-соединений. В таких случаях может быть оправданным отключение учета состояния представлений для всей страницы.

Если идея отключения учета представлений для всего файла \*.aspx кажется вам слишком радикальной, вспомните о том, что каждый потомок базового класса System.Web.UI.Control наследует свойство EnableViewState, с помощью которого очень просто отключить учет состояния представлений для каждого конкретного элемента управления в отдельности.

‹asp:GridView id="myHugeDynamicallyFilledDataGrid" runat="server" **EnableViewState="false"**›

‹/asp:GridView›

**Замечание.**Страницы ASP.NET резервируют небольшую часть строки \_\_VIEWSTATE для внутреннего использования. Именно поэтому поле \_\_VIEWSTATE появляется в браузере клиента даже тогда, когда учет состояния представлений отключен для всей страницы (или для всех элементов управления).

**Добавление пользовательских данных состояния представлений**

В дополнение к свойству EinableViewState базовый класс System.Web. UI.Control предлагает наследуемое свойство ViewState. Это свойство в фоновом режиме обеспечивает доступ к типу System.Web.UI.StateBag, представляющему все данные поля \_\_VIEWSTATE. С помощью индексатора типа StateBag вы можете встроить пользовательскую информацию в скрытое поле \_\_VIEWSTATE формы, используя для этого подходящий набор пар имён и значений. Вот простой пример.

protected void btnAddToVS\_Click(object sender, EventArgs е) {

 ViewState["CustomViewStateItem"] = "Пользовательские данные";

 lblVSValue.Text = (string)ViewState["CustomViewStateItem"];

}

Тип System.Web.UI.StateBag может работать с любым типом, производным от System.Object. Поэтому, чтобы получить доступ к значению данного ключа, вам нужно явно преобразовать его в правильный тип данных (в данном случае это тип System.String). При этом следует учесть, что размещаемое в поле \_\_VIEWSTATE значение не может быть *абсолютно любым*объектом. Единственными действительными типами в данном случае являются строки, цельте числа, булевы значения, типы ArrayList, и Hashtable, а также массивы этих типов.

Но если страницы \*.aspx могут вставлять пользовательские фрагменты информации в строку \_\_VIEWSTATE, было бы неплохо выяснить, как это сделать. Как правило, пользовательские данные состояния представлений используются для установки настроек пользователя. Например, вы можете создать элемент данных представления, указывающий то, каким пользователь желает видеть GridView (например, с точки зрения порядка сортировки). Но данные состояния представлений не очень хорошо подходят для "развёрнутых" пользовательских данных, таких как объекты в корзине покупателя, помещенные в кэш типы DataSet или какие-то другие специальные типы. Когда требуется запомнить сложную информацию, лучше работать с сеансовыми данными. Но перед тем как перейти к соответствующему разделу нашего обсуждения, мы должны выяснить роль файла Global.asax.

**Исходный код**. Файлы примера ViewStateApp размещены в подкаталоге, соответствующем главе 24.

**Несколько слов о данных состояния элементов**

В ASP.NET 2.0 предлагается поддержка *состояний элементов управления,*а не только состояния представлений. Эта технология оказывается очень удобной при работе с созданными вами Web-элементами управления ASP.NET, которые должны сохранять свои данные между последовательными циклами запросов. Для этого, конечно, можно использовать и свойство ViewState, но если учет состояния представлений будет отключен на уровне страницы, пользовательский элемент управления работать не будет. Именно по этой причине Web-элементы управления теперь поддерживают свойства ControlState.

Состояния элементов управления используются аналогично состояниям представлений, но учет состояния элементов управления не отключается при отключении учета состояния представлений на уровне страницы. Как уже говорилось, эта возможность оказывается наиболее полезной для разработчиков пользовательских Web-элементов управления (обсуждение данной темы в нашей книге не предполагается). За подробностями обратитесь к документации .NET Framework 2.0 SDK.

**Роль файла Global.asax**

К этому моменту у вас может сложиться впечатление, что приложение ASP.NET представляет собой простой набор файлов \*.аsрх и соответствующих Web-элементов управления. И хотя вы можете построить Web-приложение с помощью простого связывания нескольких Web-страниц, вам, скорее всего, хотелось бы иметь возможность взаимодействия с Web-приложением, как с чем-то цельным. Для этого в Web-приложение ASP.NET можно включить необязательный файл Global.asax, выбрав WebSite→Add New Item из меню (рис. 24.2).

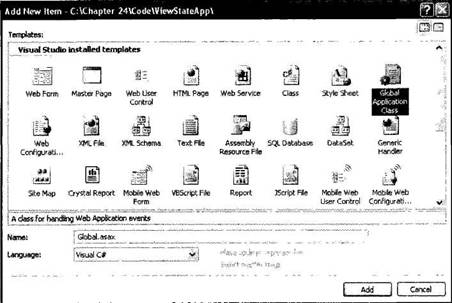


Рис. 24.2. Добавление файла Global.asax

В некотором смысле файл Global.asax в ASP.NET аналогичен традиционному файлу \*.exe, поскольку представляет поведение Web-узла в среде выполнения. После добавления файла Global.asax в Web-проект вы обнаружите, что этот файл на самом деле представляет собой блок ‹script›, содержащий набор обработчиков событий.

‹%@ Application Language="C#" %›

‹script runat="server"›

void Application\_Start(Object sender, EventArgs e) {

**// Код, выполняемый при запуске приложения.**

}

void Application\_End(Object sender, EventArgs e) {

**// Код, выполняемый при завершении приложения.**

}

void Applicatioт\_Error(Object sender, EventArgs e) {

**// Код, выполняемый при появлении необработанной ошибки.**

}

void Session\_Start(Object sender, EventArgs e) {

**// Код, выполняемый при открытии нового сеанса.**

}

void Session\_End(Object sender, EventArgs e) {

**// Код, выполняемый при завершении сеанса.**

}

‹/script›

Однако первое впечатление может оказаться обманчивым. В среде выполнении программный код этого блока ‹script› преобразуется в тип класса, получающийся из System.Web.HttpApplication. Если вы имеете опыт работы с ASP.NET 1.х, то вы, наверное, вспомните, что там файл с внешним кодом поддержки для Global.asax буквально определял класс, получающийся из HttpApplication.

Как уже было сказано, *члены,*определённые в Global.asax, содержатся в обработчиках событий, позволяющих взаимодействовать с событиями на уровне приложения (а также сеанса). Описания этих членов предлагаются в табл. 24.1.

**Таблица 24.1.**Обработчики событий Global.asax

|  |  |
| --- | --- |
| **Обработчик события** | **Описание** |
| Аррlication\_Start() | Вызывается только при запуске Web-приложения, поэтому генерируется только один раз за все время выполнения Web-приложения. Является идеальным местом для определения данных уровня приложения, доступных в любой точке Web-приложения |
| Application\_End() | Вызывается при завершении работы приложения, например, вследствие превышения времени ожидания для последнего пользователя или при завершении работы приложения вручную с помощью IIS |
| Session\_Start() | Вызывается при регистрации нового пользователя в приложении. Здесь можно установить параметры, связанные с конкретным пользователем |
| Session\_End() | Вызывается при завершении сеанса пользователя (обычно в результате превышения установленного времени ожидания) |
| Application\_Error() | Глобальный обработчик ошибок, который вызывается тогда, когда Web-приложение генерирует необработанное исключение |

**Последний глобальный шанс для обработки исключений**

Позвольте указать на роль обработчика событий Application\_Error(). Напомним, что страница может использовать обработчик события Error для обработки любого исключения, сгенерированного в контексте страницы и оставшегося без обработки. Обработчик Application\_Error() оказывается последним пунктом возможной обработки исключений, которые не были обработаны на уровне страницы. Как и в случае события Error на уровне страницы, вы можете получить доступ к конкретному объекту System.Exception, используя наследуемое свойство Server.

void Application\_Error(Object sender, EventArgs e) {

 Exception ex = Server.GetLastError();

 Response.Write(ex.Message);

 Server.ClearError();

}

Обработчик Application Error() является "последним шансом" обработки события для вашего Web-приложения, где вы, вместо предъявления сообщения об ошибке пользователю, можете записать соответствующую информацию в журнал регистрации событий Web-сервера, например:

‹%@ Import Namespace = "System.Diagnostics"%›

…

void Application\_Error(Object sender, EventArgs e) {

**// Запись последнего события в журнал событий.**

 Exception ex = Server.GetLastError();

 EventLog ev = new EventLog("Application");

 ev.WriteEntry(ex.Message, EventLogEntryType.Error);

 Server.ClearError();

 Response.Write("Это приложение "зависло". Извините!");

}

**Базовый класс HttpApplication**

Как уже говорилось, сценарий Global.asax динамически преобразуется в класс, который получается из базового класса System.Web.HttpApplication и обеспечивает те же функциональные возможности, что и тип System.Web.UI.Page. Описания соответствующих членов предлагаются в табл. 24.2.

**Таблица 24.2.**Ключевые члены типа System.Web.HttpApplication

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Application | Позволяет взаимодействовать с переменными уровня приложения, используя  доступный тип HttpApplicationState |
| Request | Позволяет взаимодействовать с входящим HTTP-запросом (с помощью HttpRequest) |
| Response | Позволяет взаимодействовать с HTTP-ответом (с помощью HttpResponse) |
| Server | Получает внутренний объект сервера для текущего запроса (с помощью HttpServerUtilitу) |
| Session | Позволяет взаимодействовать с переменными уровня сеанса, используя доступный  тип HttpSessionState |

**Различия между приложением и сеансом**

В ASP.NET состояние приложения учитывается экземпляром типа HttpApplicationState. Этот класс дает возможность сделать глобальную информацию доступной для всех пользователей (и всех страниц), зарегистрированных в вашем приложении ASP.NET. При этом можно не только открыть данные приложения для всех пользователей вашего узла, но и сделать так, чтобы при изменении значений уровня приложения одним из пользователей эти изменения становилось видимыми для всех остальных пользователей при следующих обращениях к серверу.

С другой стороны, данные состояния сеанса используются для сохранения информации, касающейся конкретного пользователя (например, товаров в корзине покупателя). Практически состояние сеанса пользователя представляется типом класса HttpSessionState. При регистрации нового пользователя в Web-прило-жении ASP.NET среда выполнения автоматически назначит этому пользователю новый идентификатор (ID) сеанса, время использования которого по умолчанию истекает после 20 минут неактивности пользователя. Если на вашем узле зарегистрируются 20000 пользователей, вы будете иметь 20000 отдельных объектов HttpSessionState, каждому из которых будет назначен свой уникальный идентификатор сеанса. Схема взаимосвязей между Web-приложением и Web-сеансами показана на рис. 24.3.

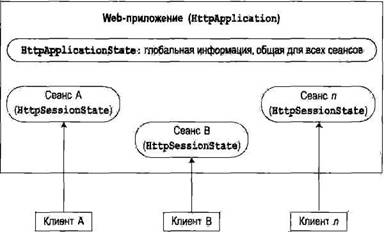


Рис. 24.3. Взаимосвязи приложения и его сеансов

Вы, возможно, знаете, что в рамках классической технологии ASP данные состояния приложения и состояния сеанса представляются разными COM-объектами (например, Application и Session). В ASP.NET типы, производные от Page. как и тип HttpApplication, используют свойства с такими же именами (т.е. Application и Session), которые предоставляют доступ к соответствующим типам HttpApplicationState и HttpSessionState.

**Поддержка данных состояния приложения**

Тип HttpApplicationState предоставляет возможность совместного использования глобальной информации для множества сеансов в приложении ASP.NET. Например, можно иметь одну строку соединения, используемую всеми страницами приложения, один общий тип DataSet, используемый множеством страниц, или любой другой фрагмент данных, доступ к которому требуется обеспечить на уровне всего приложения. Описания основных членов типа HttpApplicationState предлагаются в табл. 24.3.

**Таблица 24.3.** Члены типа HttpApplicationState

|  |  |
| --- | --- |
| **Члены** | **Описание** |
| AllKeys | Свойство, возвращающее массив типов System.String, представляющих все имена в рамках типа HttpApplicationState |
| Count | Свойство, возвращающее значение числа объектов в типе HttpApplicationState |
| Add() | Метод, позволяющий добавить новую пару "имя-значение" в тип HttpApplicationState. Этот метод используется достаточно редко, поскольку предпочтение обычно отдается индексатору класса HttpApplicationState |
| Clear() | Метод, удаляющий все элементы из типа HttpApplicationState. Функционально эквивалентен методу RemoveAll() |
| Lock() Unlock() | Эти два метода используются тогда, когда требуется изменить набор переменных приложения в реентерабельной форме |
| RemoveAll() Remove() RemoveAt() | Эти методы удаляют конкретный элемент типа HttpApplicationState (по имени строки или, как RemoveAt(), с помощью числового индексатора) |

Для создания членов-данных, которые должны быть доступны всем активным сеансам, нужно создать множество пар имен и значений. В большинстве случаев наиболее подходящим для этого местом является обработчик события Application\_Start() типа, производного от HttpApplication, например:

void Application\_Start(Object sender, EventArgs e) {

**// Установка некоторых переменных приложения.**

 Application["SalesPersonOfTheMonth"] = "Chucky";

 Application["CurrentCarOnSale"] = "Colt";

 Application["MostPopularColorOnLot"] = "черный";

}

В течение всего времени существования Web-приложения (т.е. пока Web-приложение не будет закрыто вручную или пока не истечет время ожидания последнего пользователя) любой пользователь (на любой странице) при необходимости может получить доступ к этим значениям. Предположим, что у вас есть страница, которая по щелчку мыши должна отображать в поле Label информацию об автомобиле, предлагаемом со скидкой в данный момент.

protected void btnShowCarDiscunt\_Click(object Sender, EventArgs e) {

**// Возвращаемый System.Object следует преобразовать**

**// в System.String!**

 lblCurrCarOnSale.Text = (string)Application["CurrentCarOnSale"];

}

Как и в случае свойства ViewState, обратите внимание на то, что вы должны преобразовать значение, возвращаемое типом HttpApplicationState, в подходящий тип. Поскольку тип HttpApplicationState может содержать *любой*тип, должно быть очевидно, что в рамках данных состояния приложения узла вы можете размещать пользовательские типы (и вообще любые типы .NET).

Для примера использования этой возможности создайте новое Web-приложение ASP.NET с названием AppState. Предположим, что требуется поддерживать три текущие переменные приложения в рамках строго типизованного объекта с именем CarLotlInfo.

public class CarLotInfo {

 public CarLotInfo(string s, string c, string m) {

  salesPersonOfTheMonth = s;

  currentCarOnSale = c;

  mostPopularColorOnLot = m;

 }

**// Открыты для простоты доступа.**

 public string salesPersonOfTheMonth;

 public string currentCarOnSale;

 public string mostPopularColorOnLot;

}

С этим вспомогательным классом вы можете сначала изменить обработчик события Application\_Start() так, как предлагается ниже:

protected void Application\_Start(Object sender, EventArgs e)

**// Размещение пользовательского объекта**

**// в секторе данных приложения.**

 Application["CarSiteInfo"] = new CarLotInfo("Chucky", "Colt" "черный");

}

а затем получить доступ к соответствующей информации с помощью открытых полей данных в обработчике событий сервера.

protected void btnShowAppVariables\_Click(object sender, EventArgs e) {

 CarLotInfo appVars = ((CarLotInfo)Application["CarSiteInfo"]);

 string appState = string.Format("‹li›Предлагаемая машина: [0]‹/li›", appVars.currentCarOnSale);

 appState += string.Format("‹li›Наиболее популярный цвет: {0}‹/li›", appVars.mostPopularColorOnLot);

 appState += string.Format ("‹li›Наиболее успешный продавец: {0}‹/li›", appVars.salesPersonOfTheMonth);

 lblAppVariables.Text = appState;

}

Открыв сейчас эту страницу, вы обнаружите, что в поле типа Label страницы отображаются строки каждой из переменных приложения.

**Изменение данных состояния приложения**

В ходе выполнения Web-приложения с помощью членов типа HttpApplicationState вы можете программно модифицировать или удалить любые или даже все члены уровня приложения. Например, чтобы удалить конкретный элемент, нужно просто вызвать метод Remove(). Чтобы уничтожить все данные уровня приложения, вызовите RemoveAll().

private void CleanAppData() {

**// Удаление отдельного элемента по строковому имени.**

 Application.Remove("SomeItemIDontNeed");

**// Удаление всех данных приложения.**

 Application.RemoveAll();

}

Чтобы изменить значение существующей переменной уровня приложения, нужно присвоить новое назначение соответствующему элементу данных. Предположим, что ваша страница поддерживает еще один тип Button, который дает пользователю возможность изменить имя наиболее удачливого продавца. Обработчик события Click в этом случае выглядит так, как и ожидается.

protected void btnSetNewSP\_Click(object sender, EventArgs e) {

**// Установка нового имени продавца.**

 ((CarLotInfo)Application["CarSiteInfo"]).salesPersonOfTheMonth = txtNewSP.Text;

}

Запустив это Web-приложение, вы увидите, что переменная уровня приложения обновлена. К тому же, поскольку переменные приложения доступны для всех сеансов пользователей, если запустить три или четыре экземпляра Web-браузера, вы увидите, что при изменении имени лучшего продавца одним экземпляром все остальные экземпляры браузера после вторичного обращения к серверу тоже отобразят новое значение.

Следует понимать, что в той ситуации, когда множество переменных уровня приложения должно обновляться как единое целое, есть риск нарушения целостности данных (поскольку теоретически возможно, что данные уровня приложения могут измениться в момент получения к ним доступа со стороны другого пользователя). Можно, конечно, пойти длинным путем и вручную блокировать доступ, используя примитивы потоков из пространства имен System.Threading, но тип HttpApplicationState предлагает два метода, Lock() и Unlock(), которые автоматически гарантируют потоковую безопасность.

**// Безопасный доступ к соответствующим данным приложения.**

Application.Lock();

Application["SalesPersonOfTheMonth"] = "Maxine";

Application["CurrentBonusedEmployee"] = Application["SalesPersonOfTheMonth"];

Application.Unlock();

**Замечание.**Во многом аналогично оператору lock в C#, если после вызова Lock(), но перед вызовом Unlock() возникнет исключение, блокировка будет автоматически удалена.

**Обработка завершения работы Web-приложения**

Тип HttpApplicationState предназначен для поддержки значений содержащихся в нем элементов до наступления одного из следующих событий: завершения сеанса доступа последнего пользователя узла (вследствие превышения времени ожидания или закрытия сеанса самим пользователем) или прекращения работы Web-узла вручную с помощью IIS. В любом случае будет автоматически вызван метод Application\_Exit() объекта HttpApplication. В этом обработчике события вы можете выполнять любой необходимый вам программный код "уборки".

protected void Application\_End(Object sender, EventArgs e) {

**// Запись текущих значений переменных приложения**

**// в базу данных или куда-то еще…**

}

**Исходный Код.**Файлы примера AppState размещены в подкаталоге, соответствующем главе 24

**Кэш приложения**

ASP.NET предлагает еще один, более гибкий метод обработки данных уровня приложения. Вы, конечно, помните, что значения объекта HttpApplicationState остаются в памяти до тех пор, пока работает Web-приложение. Но иногда требуется, чтобы какой-то фрагмент данных приложения поддерживался только в течение определённого ограниченного периода времени. Например, вы можете потребовать, чтобы созданный тип DataSet оставался действительным только в течение пяти минут. После этого вы предполагаете получить новый тип DataSet, учитывающий возможные пользовательские модификации. Конечно, вполне возможно построить такую структуру с применением HttpApplicationState и некоторого "ручного" управления, но соответствующая задача сильно упрощается, если использовать кэш приложения ASP.NET.

Как следует из самого названия, объект ASP.NET System.Web.Caching.Cache (доступный с помощью свойства Context.Cache) позволяет определить объект, который будет доступен всем пользователям (всех страниц) в течение определенного периода времени. В простейшей своей форме взаимодействие с объектами кэша аналогично взаимодействию с типом HttpApplicationState.

**// Добавление элемента а кэш.**

**// Этот элемент \*не\* устареет**.

Context.Cache["SomeStringItem"] = "Это строковый элемент";

string s = (string)Context.Cache["SomeStringItem"];

**Замечание**. Чтобы получить доступ к Cache из Global.asax, необходимо использовать свойство Context. Но в контексте типа, производного от System.Web.UI.Page, вы можете использовать объект Cache непосредственно.

Следует понимать, что в отсутствие необходимости автоматически обновлять (или удалять) данные уровня приложения (как в нашем случае), объект Cache не дает особых преимуществ, поскольку тогда можно непосредственно использовать тип HttpApplicationState. Но когда вы хотите, чтобы некоторые данные по прошествии определенного времени уничтожались (и, возможно, об этом сообщалось), тогда тип Cache оказывается чрезвычайно полезным.

Класс System.Web.Caching.Cache вне рамок индексатора типа определяет весьма ограниченный набор членов. Например, метод Add() можно использовать для добавления в кэш нового элемента, не определенного в данный момент (если указанный элемент уже имеется, Add() ничего не делает). Метод Insert() также помещает член в кэш. Однако если такой элемент уже определен, Insert() заменит текущий элемент новым типом. Поскольку именно это и требуется чаще всего, мы рассмотрим подробно только метод Insert().

**Кэширование данных**

Рассмотрим простой пример. Создайте новое Web-приложение ASP.NET с названием CacheState и добавьте в это приложение файл Global.asax. Подобно переменным уровня приложения, определяемым с помощью типа HttpApplicationState. тип Cache может содержать любой тип System.Object, а его значения часто задаются в обработчике событий ApplicationStart(). Для нашего примера целью является автоматическое обновление содержимого DataSet каждые 15 секунд. Этот тип DataSet будет содержать текущий набор записей из таблицы Inventory базы данных Cars, созданной нами при обсуждении ADO.NET. С учетом этого обновите свой тип класса Global так, как показано ниже (сразу же после листинга предлагается анализ этого программного кода).

‹%@ Application Language="C#" %›

‹%@ Import Namespace = "System.Data.SqlClient" %›

‹%@ Import Namespace = "System.Data" %›

‹script runat="server"›

**// Определение статического члена-переменной типа Cache.**

static Cache theCache;

void Application\_Start(Object sender, EventArgs e) {

**// Присваивание значения статической переменной 'theCache' .**

 theCache = Context.Cache;

**// При запуске приложения читается текущая запись**

**// таблицы Inventory базы данных Cars.**

 SqlConnection cn = new SqlConnection("data source=localhost;initial catalog=Cars; user id='sa';pwd=''");

 SqlDataAdapter dAdapt = new SqlDataAdapter("Select *\**From Inventory", cn);

 DataSet theCars = new dataSet(); dAdapt.Fill(theCars, "Inventory");

**// Сохранение DataSet в кэше.**

 theCache.Insert("AppDataSet", theCars, null,

  DateTime.Now.AddSeconds(15), Cache.NoSlidingExpiration, CacheItemPrioritу.Default,

  new CacheItemRemovedCallback(UpdateCarInventory));

}

**// Цель делегата CacheItemRemovedCallback.**

static void UpdateCarInventorу(string key, object item, CacheItemRemovedReason reason) {

**// Заполнение DataSet.**

 SqlConnection cn*=* new SqlConnection("data source=localhost;initial catalog=Cars; user id = 'sa';pwd=''");

 SqlDataAdapter dAdapt = new SqlDataAdapter("Select \* From Inventory", cn);

 DataSet theCars = new DataSet();

 dAdapt.Fill(theCars, "Inventory");

**// Сохранение в кэше.**

 theCache.Insert("AppDataSet", theCars, null,

  DateTime.Now.AddSeconds(15), Cache.NoSlidingExpiration, CacheItemPriority.Default,

  new CasheItemRemovedCallback(UpdateCarInventory));

}

…

‹/script›

Сначала обратите внимание на то, что тип Global определяет статическую переменную типа Cache. Причина в том, что определяется также статическая функция (UpdateCarInventory()), которой требуется доступ к Cache (напомним, что статические члены не имеют доступа к наследуемым членам, поэтому в данном случае вы не сможете использовать свойство Context).

Внутри обработчика событий Application\_Start() заполняется тип DataSet, который затем помещается в кэш приложения. Должно быть ясно, что метод Context.Cache.Insert() перегружен. Ниже объясняются значения каждого из параметров этого метода.

**// Сохранение в кэше.**

theCache.Insert("AppDataSet", **// Имя для идентификации элемента.**

 theCars,**// Объект для отправки в кэш.**

 null,**// Зависимости для объекта.**

 DateTime.Now.AddSeconds(15),**// Длительность пребывания в кэше.**

 Cache.NoSlidingExpiration,**// Фиксированное или скользящее время.**

 CacheItemPriority.Default,**// Приоритет элемента.**

**// Делегат для события CacheItemRemove.**

 new CacheItemRemovedCallback(UpdateCarInventory));

Первые два параметра просто формируют пару "имя-значение" элемента. Третий параметр позволяет определить тип Cache Dependency (который в данном случае будет равен null, поскольку у вас в кэше нет элементов, зависящих от DataSet).

**Замечание.**Возможность определить тип CacheDependency довольно привлекательна. Например, можно указать зависимость между некоторым членом и внешним файлом, и если содержимое файла изменится, соответствующий тип автоматически обновится. Все подробности можно найти в документации .NET Framework 2.0.

Следующие три параметра используются для определения приоритета элемента и времени, в течение которого элементу позволяется оставаться в кэше приложения. Здесь указывается доступное только для чтения поле Cache.NoSlidingExpiration, которое указывает, что указанный промежуток времени (15 секунд) является абсолютным. Наконец, и это самое важное для данного примера, создается тип делегата CacheItemRemovedCallback, которому передается имя метода, вызываемого при очистке DataSet. Как следует из структуры метода UpdateCarInventory(), делегат CacheItemRemovedCallback может вызвать только методы со следующей сигнатурой.

static void UpdateCarInventory(string key, object item, CacheItemRemovedReason reason) {…}

Теперь при запуске приложения тип DataSet будет заполнен и помещен в кэш. Каждые 15 секунд DataSet будет очищаться, обновляться и снова помещаться в кэш. Чтобы увидеть результат этих действий, мы должны создать тип Page, который будет позволять некоторую степень взаимодействия с пользователем.

**Изменение файла \*.aspx**

Обновите пользовательский интерфейс исходного файла \*.aspx так, как показано на рис. 24.4.

В обработчике события Load страницы настройте GridView на отображение содержимого помещенного в кэш типа DataSet при первом обращении пользователя к странице.

protected void Page\_Load(object sender, EventArgs e) {

 if (!IsPostBack) {

  carsGridView.DataSource = (DataSet)Cache["AppDataSet"];

  carsGridView.DataBind();

 }

}

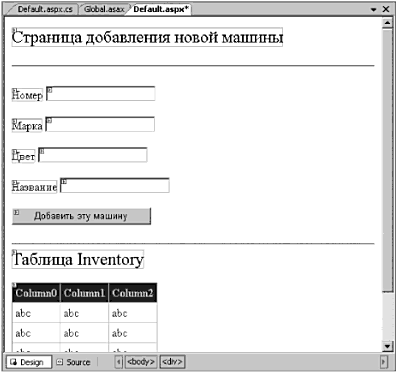


Рис. 24.4. Графический интерфейс пользователя для приложения с кэшированием

В обработчике события Click кнопки Добавить эту машину вставьте новую запись в базу данных Cars, используя для этого объект ADO.NET SqlCommand. После добавления записи вызовите вспомогательную функцию RefreshGrid(), которая обновит интерфейс с помощью типа SqlDataReader (поэтому не забудьте указать using для пространства имен System.Data.SqlClient). Вот как должны выглядеть соответствующие методы,

protected void btnAddCar\_Click(object sender, EventArgs e) {

**// Обновление таблицы Inventory**

**// и вызов RefreshGrid**().

 SqlConnection cn = new SqlConnection();

 cn.ConnectionString = "User ID=sa;Pwd=;Initial Catalog=Cars;Data Source=(local)";

 cn.Open();

 string sql;

 SqlCommand cmd;

**// Вставка нового Car.**

 sql = string.Format(

  "INSERT INTO Inventory(CarID, Make, Color, PetName) VALUES" +

  "('{0}', '{1}', '{2} ', '{3}')",

  txtCarID.Text, txtCarMake.Text,

  txtCarColor.Text, txtCarPetName.Text);

 cmd = new SqlCommand(sql, cn);

 cmd.ExecuteNonQuery();

 cn.Close();

 RefreshGrid();

}

private void RefreshGrid() {

**// Заполнение таблицы.**

 SqlConnection cn = new SqlConnection();

 cn.ConnectionString = "User ID=sa;Pwd=;Initial Catalog=Cars;Data Source=(local)";

 cn.Open();

 SqlCommand cmd = new SqlCommand("Select \* from Inventory", cn);

 carsGridView.DataSource = cmd.ExecuteReader();

 carsGridView.DataBind();

 cn.Close();

}

Теперь, чтобы проверить использование кэша, запустите два экземпляра вашего Web-браузера и перейдите к этой странице \*.aspx. Вы должны увидеть, что оба типа DataGrid отображают одинаковую информацию. В окне одного из экземпляров браузера добавьте новую машину. Очевидно, что в результате вы увидите обновленные данные GridView в окне браузера, который инициировал обращение к источнику данных.

Во втором экземпляре браузера щелкните на кнопке Обновить. Вы *не*увидите новый элемент, поскольку обработчик события Page\_Load читает данные непосредственно из кэша. (Если же вы увидели новые данные, это значит, что уже истекли 15 секунд. Либо печатайте быстрее, либо увеличьте время, в течение которого тип DataSet должен оставаться в кэше.) Подождите несколько секунд и снова щелкните на кнопке Обновить второго экземпляра браузера. Теперь вы *должны*увидеть новые данные, поскольку время пребывания DataSet в кэше истекло, и целевой метод делегата CacheItemRemovedCallback автоматически обновил тип DataSet, помещенный в кэш.

Как видите, главное преимущество типа Cache заключается в том, что вы получаете возможность ответить на удаление члена. В этом примере вы, конечно, можете избежать использования типа Cache путем чтения данных в обработчике Page\_Load() непосредственно из базы данных Cars. Однако теперь вам должно быть ясно, что кэш позволяет автоматически обновлять данные с помощью делегатов .NET.

**Замечание.**В отличие от типа HttpApplicationState, класс Cache не поддерживает методы Lock() и Unlock(). Так что при необходимости обновить связанные элементы вам придется использовать типы из пространства имен System.Threading или ключевое слово lock C#.

**Исходный код.**Файлы примера CacheState размещены в подкаталоге, соответствующем главе 24.

**Обработка сеансовых данных**

Поговорив о данных уровня приложения, давайте перейдем к обсуждению данных, создаваемых на уровне пользователя. Как уже упоминалось, *сеанс*на самом деде представляет собой процесс взаимодействие пользователя с Web-приложением, представленный типом HttpSessionState. Для поддержки информации сеанса конкретного пользователя объект HttpApplication и любые другие типы System.Web.UI.Page могут использовать доступ к свойству Session. Классическим примером необходимости поддержки пользовательских данных является корзина покупателя: при подключении десятков посетителей к странице Интернет-магазина для каждого посетителя должен поддерживаться уникальный список товаров, которые этот посетитель собрался купить.

При регистрации нового пользователя в Web-приложении среда выполнения .NET автоматически назначит пользователю уникальный идентификатор сеанса, используемый для идентификации данного пользователя. С каждым идентификатором сеанса ассоциируется пользовательский экземпляр типа HttpSessionState, который будет содержать данные соответствующего пользователя. Технология добавления и чтения сеансовых данных синтаксически идентична работе c данными приложения, например:

**// Добавление/чтение сеансовой переменной для данного пользователя.**

Session["DesiredCarColor"] = "зеленый";

String color = (string)Session["DesiredCarColor"];

Производный от HttpApplication тип позволяет выполнить перехват событий начала и завершения сеанса с помощью обработчиков событий Session\_Start() и Session\_End(). В пределах Session\_Start() вы можете создать любые элементы данных пользователя, а в Session\_End() можно выполнить любые действия, необходимые при завершения сеанса пользователя.

‹%@ Application Language="C#" %*›*

‹script runat="server"›

…

void Session\_Start(objecl sender, EventArgs e) {}

void Session\_End(object sender, EventArgs e) {}

‹/script›

Подобно типу HttpApplicationState, тип HttpSessionState может содержать любой тип, производный от System.Object, включая пользовательские классы. Предположим, например, что у нас есть новое Web-приложение (SessionState), которое определяет вспомогательный класс с именем UserShoppingCart.

public class UserShoppingCart {

 public string desiredCar;

 public string desiredCarColor;

 public float downPayment;

 public bool isLessing;

 public DateTime dateOfPickUp;

 public override string ToString() {

  return string.Format("Машина: {0}‹br›Цвет: {1}‹br›$ кредит: {2}‹br›" +

   "Аренда: {3}‹br›Доставка: {4}",

   desiredCar, desiredCarColor, downPayment, isLeasing,

   dateOfPickUp.ToShortDateString());

 }

}

В обработчике событий Session\_Start() можно назначить каждому пользователю свой экземпляр класса UserShoppingCart.

void Session\_Start(Object sender, EventArgs e) {

 Session["UserShoppingCartInfo"] = new UserShoppingCart();

}

При просмотре ваших Web-страниц пользователем вы можете взять экземпляр UserShoppingCart и заполнить его поля данными соответствующего пользователя. Предположим, что у вас есть простая страница \*.aspx с набором элементов ввода, соответствующих каждому полю типа UserShoppingCart, и кнопкой (Button), используемой для установки введенных значений (рис. 24.5).

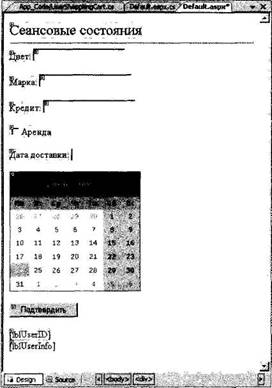


Рис. 24.5. Графический интерфейс пользователя для приложения с сеансовыми данными

Серверный обработчик события Click действует весьма прямолинейно (считывает значения элементов TextBox и отображает поступающие значения в поле типа Label).

protected void btnSubmit\_Click(object sender, EventArgs e) {

**// Установка преференций текущего пользователя.**

 UserShoppingCart u = (UserShoppingCart)Session["UserShoppingCartInfo"];

 u.DateOfPickUp = myCalendar.SelectedDate;

 u.desiredCar = txtCarMake.Text;

 u.desiredCarColor = txtCarColor.Text;

 u.downPayment = float.Parse(txtDownPayment.Text);

 u.isLeasing = chkisLeasing.Checked;

 lblUserInfo.Text = u.ToString();

 Session["UserShoppingCartInfo"] = u;

}

В Session\_End() вы можете, например, сохранить значения полей UserShoppingCart в базе данных или выполнить какие-то иные действия. Так или иначе, если Вы запустите два или три экземпляра своего браузера, вы должны увидеть, что каждый пользователь может создать свою корзину покупателя, связанную с его уникальным экземпляром HttpSessionState.

**Дополнительные члены HttpSessionState**

Кроме индексатора типа, класс HttpSessionState определяет ряд других интересных членов. Во-первых, свойство SessionID возвращает уникальный идентификатор текущего пользователя.

lblUserID.Text = string.Format("Значение вашего ID: {0}", Session.SessionID);

Методы Remove() и RemoveAll() можно использовать для удаления элементов из экземпляра HttpSessionState пользователя.

Session.Remove["НекоторыеУжеНенужныеЭлементы"];

Тип HttpSessionState определяет также набор членов, управляющих значениями времени ожидания для текущего сеанса. Снова подчеркнем, что по умолчанию каждому пользователю позволяется 20 минут бездействия до того, как объект HttpSessinState будет уничтожен. Поэтому если пользователь войдет в ваше Web-приложение (и получит в результате этого свое уникальное значение идентификатора сеанса), но не будет обращаться к узлу в течение 20 минут, среда выполнения "решит", что пользователь больше не интересуется узлом и уничтожает все сеансовые данные этого пользователя. Вы имеете возможность изменить это принятое по умолчанию 20-минутное значение для каждого пользователя в отдельности, используя свойство Timeout. Чаще всего для такого изменения используется контекст метода Global.Session\_Start().

protected void Session\_Start(Object sender, EventArgs e) {

**// Пользователю разрешается 5 минут бездействия.**

 Session.Timeout = 5;

 Session["UserShoppingCartInfo"] = new UserShoppingCart();

}

**Замечание.**Чтобы не менять значение Timeout каждого пользователя, вы можете изменить принятое по умолчанию 20-минутное значение для всех пользователей сразу с помощью атрибута Timeout элемента ‹sessionState› в файле Web.config (структура и возможности этого файла будут рассмотрены в конце главы).

Преимущество использования свойства Timeout заключается в том, что вы можете назначить свои значения времени ожидания для каждого пользователя в отдельности. Например, представьте себе, что ваше Web-приложение позволяет пользователям заплатить за некоторый уровень членства. Вы можете потребовать, чтобы "золотым" членам устанавливалось время ожидания, равное одному часу, а "деревянным" – только 30 секунд. Такая возможность порождает следующий вопрос: как реализовать запоминание соответствующей информации пользователя (например, текущий уровень его членства) между обращениями к Web-странице? Одной из возможностей является использование типа HttpCookie.

**Исходный код,**Файлы примера SessionState размещены в подкаталоге, соответствующем главе 24.

**Данные cookie**

Последней из рассмотренных здесь технологий управления данными состояния будет использование данных *cookie,*которые часто имеют вид обычных текстовых файлов (или наборов файлов), сохраняемых на машине пользователя. При регистрации пользователя данного узла браузер проверяет, есть ли на машине пользователя файл cookie для данного URL, и если такой файл обнаруживается, данные файла присоединяются к HTTP-запросу.

Получающая запрос Web-страница на сервере может прочитать данные cookie, чтобы использовать их при создании графического интерфейса, учитывающего текущие предпочтения пользователя. Уверен, при посещении своих любимых Web-узлов вы замечали, что узел как будто знает, какого сорта содержимое вы хотите видеть. Например, при регистрации на странице http://www.ministryofsound.com мне автоматически предъявляется содержимое, отвечающее моим музыкальным вкусам. Причиной (отчасти) является то, что на моем компьютере были сохранены данные cookie с информацией о том типе музыки, которую я предпочитаю слушать.

Точное место хранения файлов cookie зависит от используемого вами браузера. При использовании Microsoft Internet Explorer файлы cookie по умолчанию сохраняются в папке C:\Documents and Sеttinngs\‹имяПользователя›\Cookies (рис. 24.6).

Содержимое конкретного файла cookie, очевидно, будет зависеть от URL, но это, в конечном счете, обычные текстовые файлы. Поэтому вариант использования данных cookie нельзя считать удачным для передачи конфиденциальной информации о текущем пользователе (например, номера кредитной карточки, пароля или другой аналогичной информации). Даже если данные будут зашифрованы, какой-нибудь хакер может расшифровать соответствующие значения и использовать их в злонамеренных целях. Но, так или иначе, файлы cookie играют важную роль в разработке Web-приложений, поэтому нам важно выяснить, как эта специфическая технология управления состоянием отражается в ASP.NET.

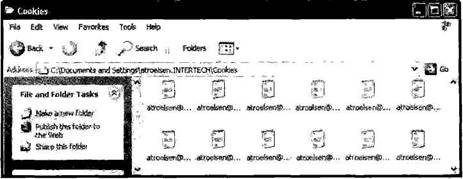


Рис. 24.6. Данные cookie, сохраненные браузером Microsoft Internet Explorer

**Создание данных cookie**

Во-первых, важно понять, что в ASP.NET данные cookie могут быть перманентными или временными, *Перманентные* данные cookie Обычно рассматриваются в смысле классического определения данных cookie, т.е. как множество пар имен и значений, физически сохраненных на жестком диске пользователя. *Временные*данные cookie (которые также называются сеансовыми данными cookie) содержат те же данные, что и перманентные cookie, но в этом случае пары имен и значений не сохраняются на машине пользователя, а существуют *только*в пределах заголовка HTTP-сообщения. При отключении пользователя от вашего узла все данные сеансовых cookie уничтожаются.

**Замечание.**Большинство браузеров поддерживает строки cookie длиной до 4096 байт. Из-за такого ограничения строки cookie лучше всего использовать для запоминания небольших фрагментов данных, например идентификаторов пользователей, с помощью которых затем можно получить дополнительные данные из базы данных.

Тип System.Web.HttpCookie является классом, представляющим серверную часть данных cookie (перманентных или временных). Для создания новых данных cookie используется свойство Response.Cookies. После вставки нового объекта HttpCookie во внутреннюю коллекцию) соответствующие пары имен и значений направляются обратно браузеру в пределах заголовка HTTP.

Для примера использования данных cookie создайте новое Web-приложение ASP.NET (CookieStateApp) с пользовательским интерфейсом, изображенным на рис. 24.7.

В обработчике события Сlick кнопки создайте новый тип HttpCookie и вставьте его в коллекцию Cookie, доступную с помощью свойства HttpRequest.Cookies.

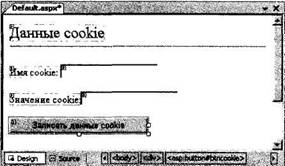


Рис. 24.7. Пользовательский интерфейс приложения CookieStateApp

Следует знать о том, что данные cookie не будут сохраняться на жестком диске пользователя, если вы не укажете явно дату истечения срока действия этих данных, используя свойство HttpCookie.Expires. Поэтому в следующем фрагменте программного кода создаются временные данные cookie, которые будут уничтожены, когда пользователь прекратит работу браузера.

protected void btnInsertCookie\_Click (object sender, EventArgs e) {

**// Создание новых (временных) данных cookie.**

 HttpCookie theCookie = new HttpCookie(txtCookieName.Text, txtCookieValue.Text);

 Response.Cookies.Add(theCookie);

}

Однако следующий фрагмент программного кода генерирует перманентные данные cookie, срок действия которых истечет 24 марта 2009 года.

private void btnInsertCookie\_Click(object sender, EventArgs e) {

**// Создание новых (перманентных) данных cookie.**

 HttpCookie theCookie = new HttpCookie(txtCookieName.Text, txtCookieValue.Text);

 theCookie.Expires = DateTime.Parse("03/24/2009");

 Response.Cookies.Add(theCookie);

}

Запустив это приложение и добавив некоторые данные cookie, вы сможете проверить, что браузер автоматически сохранит эти данные на диске. Открыв соответствующий текстовый файл, вы увидите нечто, подобное показанному на рис. 24.8.

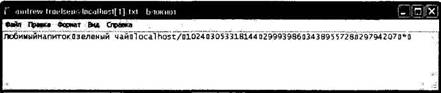


Рис. 24.8. Перманентные данные cookie

**Чтение поступающих данных cookie**

Напомним, что именно браузер отвечает за возможность доступа к перманентным данным cookie во время обращения к ранее посещавшейся странице. Для взаимодействия с поступающими данными cookie в ASP.NET предусмотрено свойство HttpRequest.Cookies. Например, если вы хотите обновить пользовательский интерфейс вашего приложения с тем, чтобы можно было отобразить текущие данные cookie с помощью элемента управления Button, вы можете реализовать цикл по всем парам имея и значений, представляя соответствующую информацию в поле элемента Label.

protected void btnShowCookies\_Click(object sender, EventArgs e) {

 string cookieData = "";

 foreach(string s in Request.Cookies) {

  cookieData += string.format("‹li›‹b›Имя‹/b›: {0}, ‹b›Значение‹/b›: {1}‹/li›", s, Request.Cookies[s].Value);

 }

 lblCookieData.Text = cookieData;

}

Если теперь запустить приложение и щелкнуть на новой кнопке, вы увидите, что данные cookie вашим браузером действительно посылаются (рис. 24.9).

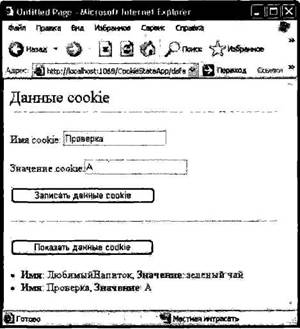


Рис. 24.9. Просмотр данных cookie

К этому моменту мы с вами рассмотрели множество способов запоминания информации о пользователях. Вы видели, что данные состояния представлений и данные приложения, кэша, сеанса и cookie обрабатываются примерно одинаково (с помощью индексатора класса). Вы также видели, что тип HttpApplication часто используется для перехвата и обработки событий, происходящих в течение всего времени существования Web-приложения. Следующей нашей задачей является выяснение роли файла Web.config.

**Исходный код.**Файлы примера CookieStateApp размещены в подкаталоге, соответствующем главе 24.

**Настройка Web-приложения ASP.NET с помощью Web.config**

При изучении компоновочных блоков .NET мы с вами выяснили, что приложения клиента могут использовать XML-файл конфигурации, содержащий инструкции CLR о том, как обрабатывать связанные запросы, где искать необходимые компоновочные блоки и что еще нужно учесть в среде выполнения. То же можно сказать и в случае Web-приложений ASP.NET, но в данном случае файлы конфигурации (впервые упомянутые в главе 23) всегда называются Web.config (в отличие от файлов конфигурации \*.exe, имена которых зависят от имен соответствующих выполняемых файлов клиента).

При добавлении файла Web.config к файлам узла с помощью выбора WebSite→Add New Item из меню создаваемая по умолчанию структура выглядит примерно так, как показано ниже (чтобы не загромождать структуру, комментарии здесь были исключены).

‹?xml version="1.0"?›

 ‹configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0"›

  ‹appSettings/›

  ‹connectionStrings/›

  ‹system.web›

   ‹compilation debug="false"/›

   ‹authentication mode="Windows"/›

  ‹/system.web›

‹/configuration›

Подобно любому файлу \*.config, в файле Web.config определяется корневой элемент ‹configuration›. В его контекст вкладывается элемент ‹system.web›, который может содержать множество дочерних элементов, с помощью которых осуществляется управление поведением Web-приложения в среде выполнения. В ASP.NET файл Web.config можно модифицировать с помощью любого текстового редактора. Некоторые элементы, которым позволено присутствовать в файле Web.config, описаны в табл. 24.4.

**Замечание.**Чтобы выяснить подробности формата файла Web.config, выполните поиск разделов документации .NET Framework 2.0 SDK, соответствующих ключу поиска "ASP.NET Settings Schema".

**Таблица 24.4.**Подборка элементов файла

|  |  |
| --- | --- |
| **Элемент** | **Описание** |
| ‹appSettings*›* | Используется для создания пользовательских пар имен и значений, которые можно программно считывать в память для использования на страницах в дальнейшем |
| ‹authentication› | Связанный с безопасностью элемент, используемый для определения режима аутентификации данного Web-приложения |
| ‹authorization› | Еще один связанный с безопасностью элемент, используемый для определения прав пользователей при доступе к ресурсам сервера |
| ‹compilation› | Используется для разрешения (или запрета) отладки и определения языка .NET, используемого данным Web-приложением по умолчанию, а также (необязательно) для определения множества внешних компоновочных блоков .NET ссылки на которые должны использоваться автоматически |
| <connectionStrings> | Используется для хранения строк внешних соединений данного Web-узла |
| ‹customErrors› | Используется для инструкций среде выполнения по поводу того, как сообщать об ошибках, происходящих в процессе работы Web-приложения |
| ‹globalization› | Используется для настройки параметров глобализации данного Web-приложения |
| ‹sessionState› | Используется для контроля того, как и где среда выполнения .NET должна хранить данные состояния сеанса |
| <trace> | Используется для разрешения (или отключения) трассировки данного Web-приложения |

Файл Web.config может содержать дополнительные элементы, размещенные как до, так и после элементов, представленных в табл. 24.4. Большинство этих эле-ментов связано с безопасностью, а остальные оказываются полезными только для построении достаточно сложных сценариев ASP.NET, предполагающих, например, создание пользовательских HTTP-заголовков или пользовательских HTTP-модулей (эти вопросы здесь обсуждать не планируется). Если вам нужен полный комплект элементов, допустимых для использования в файле Web.config, поищите по ключу "ASP.NET Settings Schema" в системе оперативной справки.

**Разрешение трассировки с помощью ‹trace›**

Первым элементом файла Web.config, который мы собираемся здесь рассмотреть, будет элемент ‹trace›. Этот XML-дескриптор может иметь любое число атрибутов, задающих особенности его поведения, как показано в следующем примере.

‹trace

 enabled="true|false"

 localOnly= "true|false"

 pageOutput="true|false"

 requestLimit="integer"

 traceMode="SortByTima|SortByCategory"/›

Описания этих атрибутов предлагаются в табл. 24.5.

**Таблица 24.5.**Атрибуты элемента ‹trace›

|  |  |
| --- | --- |
| **Атрибут** | **Описание** |
| enabled | Индикатор разрешения трассировки для приложения в целом (значением по умолчанию является false – ложь). Как было показано в предыдущей главе, можно разрешить трассировку селективно для данного файла \*.aspx, используя директиву @Page |
| localOnly | Индикатор отображения информации трассировки только на Web-сервере, но не в системах удаленных клиентов (значением по умолчанию является true – истина) |
| pageOutput | Указывает вид представления результатов трассировки |
| requestLimit | Указывает число запросов трассировки, сохраняемых на сервере. Значением по умолчанию является 10. Если достигается предел, трассировка автоматически отключается |
| traceMode | Указывает соответствие порядка отображения информации трассировки порядку ее обработки. Значением по умолчанию является SortByTime (сортировка по времени), но можно также указать сортировку по категории |

Вспомните из предыдущей главы, что с помощью директивы ‹%@Page%› можно разрешить трассировку отдельных страниц. Но если вы хотите разрешить трассировку для всех страниц Web-приложения, измените ‹trace› в файле Web.config, как показано ниже.

‹trace **enabled="true**" requestLimit="10" pageOutput="false" traceMode="SortByTime" localOnly="true" /›

**Настройка вывода сообщений об ошибках с помощью ‹customErrors›**

Элемент ‹customErrors› может использоваться для автоматического перенаправления всех ошибок в пользовательский набор файлов \*.htm. Это может оказаться полезным тогда, когда вы хотите построить более понятную для пользователя страницу информирования об ошибках по сравнению с той, которая предлагается средой CLR по умолчанию. В общем виде элемент ‹customErrors› выглядит так.

‹customErrors defaultRedirect="url" mode="On|Off|RemoteOnly"›

 ‹error statusCode="код\_состояния" redirect="url"/›

‹/customErrors›

Чтобы предъявить пример применения элемента ‹customErrors›, предположим, что Web-приложение ASP.NET имеет два файла \*.htm. Первый файл (genericError.htm) функционирует, как страница перехвата ошибок. Эта страница может содержать логотип вашей компании, ссылку на адрес электронной почты администратора системы и, например, сообщение с извинениями за доставленные пользователю неудобства. Второй файл (Error404.htm) – это пользовательская страница ошибки, которая должна появляться только тогда, когда среда выполнения обнаруживает ошибку с номером 404 (грозная ошибка "необнаруженного ресурса"). Если вы хотите, чтобы все ошибки обрабатывались этими пользовательскими страницами, вы можете изменить файл Web.config так, как предлагается ниже.

‹?xml version="1.0"?›

‹configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0"›

 ‹appSettings/›

 ‹connectionStrings /›

 ‹system.web›

  ‹compilation debug= "false" /›

  ‹authentication mode="Windows"/›

**‹customErrors defaultRedirect="genericError.htm" mode="On"›**

**‹error statusCodes="404" redirect="Error404.htm"/›**

**‹/customErrors›**

 ‹/system.web›

‹/configuration›

Обратите внимание на то, что корневой элемент ‹customErrors› используется для указания имени общей страницы для всех необработанных ошибок. Одним из атрибутов, которые могут присутствовать в открывающем дескрипторе, является атрибут mode. Значением, устанавливаемым для этого атрибута по умолчанию, является RemoteOnly, дающее указание среде выполнения *не отображать*пользовательские страницы ошибок, если HTTP-запрос поступает с той же машины, где находится Web-cервер (это очень удобно для разработчиков, которым требуется видеть все подробности). Если установить для атрибута mode значение "on", это позволит видеть пользовательские ошибки на всех машинах (включая машину разработки). Также заметьте, что элемент ‹customErrors› может поддерживать любое число вложенных элементов ‹error›, с помощью которых можно указать, какая страница должна иcпользоваться для ошибки с тем или иным кодом.

Чтобы проверить работу пользовательского перенаправления ошибок, создайте страницу \*.aspx с двумя элементами управления Button и обработайте их события Click так, как предлагается ниже.

private void btnGeneralError\_Click(object sender, EventArgs e) {

**// Это генерирует ошибку общего вида.**

 throw new Exception("Ошибка общего вида…");

}

private void btn404Error\_Click(object sender, EventArgs e) {

**// Это генерирует ошибку 404 (при отсутствии файла MyPage.aspx).**

 Response.Redirect("MyPage.aspx");

}

**Сохранение данных состояния с помощью ‹sessionState›**

Наиболее мощным элементом файла Web.config является ‹sessionState›. По умолчанию ASP.NET запоминает данные сеансового состояния с помощью \*.dll в рамках рабочего процесса ASP.NET (aspnet\_wp.exe). Подобно любому файлу \*.dll. положительным моментом его использования является то, что доступ к информации оказывается настолько быстрым, насколько это возможно. Однако недостатком оказывается то, что в случае аварийного завершения работы этого домена приложения (по любой причине) будут потеряны все данные пользователя. К тому же, когда вы храните данные в \*.dll внутри процесса, вы не можете взаимодействовать с сетевой Web-группой. По умолчанию элемент ‹sessionState› файла Web.config выглядит примерно так.

‹sessionState **mode="InProc"**stateConnectionString="tcpip=127.0.0.1:42424" sqlConnectionString="data source=127.0.0.1;Trusted\_Connection=yes" cookieless="false" timeout="20" /›

Принятый по умолчанию режим хранения оказывается вполне подходящим только в том случае, когда ваше приложение обслуживается в рамках одного Web-сервера. Но в ASP.NET вы можете дать указание среде выполнения обслуживать \*.dll сеансового состояния в суррогатном процессе, называемом сервером сеансового состояния ASP.NET (aspnet\_state.exe). Тем самым вы можете вывести \*.dll из aspnet\_wp.exe в отдельный \*.exe. Первым шагом при этом должен быть запуск службы aspnet\_state.exe Windows. С этой целью введите в командной строке

net start aspnet\_state

Запустить aspnet\_state.exe можно и по-другому, с помощью оснастки Службы, доступной из папки Администрирование панели управления Windows (рис. 24.10).

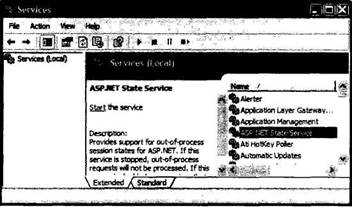


Рис. 24.10. Оснастка Services Windows

Преимуществом этого подхода является то, что с помощью окна свойств вы можете настроить aspnet\_state.exe на автоматический старт при загрузке машины. В любом случае, запустив сервер состояния сеанса, измените элемент ‹sessionState› в файле Web.config так, как показано ниже.

‹sessionState **mode="StateServer"**stateConnectionString="tcpip=127.0.0.1:42424" sqlConnectionString="data source=127.0.0.1;Trusted\_Connection=yes" cookieless="false" timeout="20"/›

Здесь атрибут mode устанавливается равным StateServer. Это именно то, что требуется. Теперь CLR обслуживает данные сеанса в рамках aspnet\_state.exe. В этом случае, если домен приложения, содержащий само Web-приложение, завершится аварийно, сеансовые данные сохранятся. Также заметьте, что элемент ‹sessionState› может содержать атрибут stateConnectionString. Принятое по умолчанию значение (127.0.0.1) для адреса TCP/IP указывает на локальную машину. Если вместо этого вы хотите, чтобы среда выполнения .NET использовала сервис aspnet\_state.exe на другой машине в сети (снова подумайте о Web-группе), вы имеете возможность изменить это значение.

Наконец, если требуется наивысшая степень изолированности и устойчивости дли Web-приложения, вы можете "заставить" среду выполнения сохранять все данные состояния сеанса в Microsoft SQL Server. Соответствующая модификация файла Web.config снова очень проста.

‹sessionState **mode="SQLServer"** stateConnectionString="tcpip=l27.0.0.1:42424" sqlConnectionString="data sourse=127.0.0.1;Trusted\_Connection=yes'' cookieless="false" timeout="20" /›

Однако перед тем как вы попытаетесь выполнять соответствующее Web-приложение, вы должны обеспечить правильную настройку целевой машины (указанной атрибутом sqlConnectionString). При установке .NET Framework 2.0 SDK (или Visual Studio 2005) создаются два файла, InstallSqlState.sql и UninstallSqlScate.sql, которые по умолчанию помещаются в папку ‹%windir%›\Microsoft.NET\Framework\‹версия›. На целевой машине вы должны выполнить файл InstallSqlState.sql, используя, например, SQL Server Query Analyzer (который входит в поставку Microsoft SQL Server).

После выполнения указанного SQL-сценария, вы обнаружите новую базу данных SQL Server (с именем ASPState), содержащую набор хранимых процедур, вызываемых средой выполнения ASP.NET, и множество таблиц, используемых для хранения сеансовых данных (кроме того, в базу данных tempdb будет добавлено множество таблиц для обмена данными). Вы должны понимать, что настройка Web-приложения на запоминание сеансовых данных рамках SQL Server является самым медленным из всех возможных вариантов. Преимущество этого варианта в том, что пользовательские данные при этом сохраняются наиболее надежно (даже при перезапуске Web-сервера).

**Замечание.**Если для хранения сеансовых данных используется сервер состояния сеанса ASP.NET или SQL Server, то любой пользовательский тип, размещаемый в объекте HttpSessionState, должен быть обозначен атрибутом [Serializable].

**Утилита администрирования узла ASP.NET 2.0**

В завершение этого раздела главы следует упомянуть тот факт, что ASP.NET 2.0 теперь предлагает Web-утилиту конфигурации для управления множеством установок в файле Web.config узла. Чтобы активизировать эту утилиту (рис. 24.11), выберите Web Site→ASP.NET Configuration из меню Visual Studio 2005.

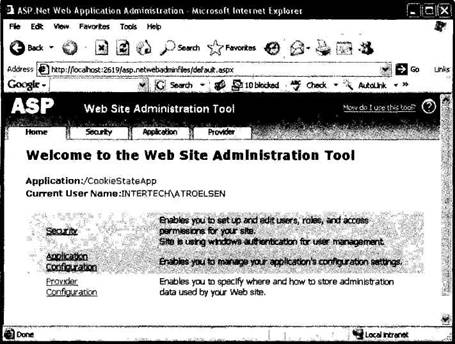


Рис. 24.11. Утилита администрирования узла ASP.NET 2.0

В большинстве своем функциональные возможности этого инструмента связаны с безопасностью вашего узла (режим аутентификации, пользовательские роли, поставщики безопасности и т.д.). Однако, кроме того, этот инструмент позволяет устанавливать параметры приложения, детали отладки и страницы обработки ошибок.

**Наследование конфигурации**

Последним из рассматриваемых в этой главе вопросов будет *наследование конфигурации.*Из предыдущей главы вы узнали, что Web-приложение можно определить, как множество файлов, содержащихся в корневом каталоге *и любом числе необязательных подкаталогов*. Все приложения примеров этой и предыдущей глав находились в одном корневом каталоге, управляемом IIS (с необязательным каталогом Bin). Однако полномасштабные Web-приложения обычно определяют в корневом каталоге множества подкаталогов, каждый из которых содержит некоторое подмножество связанных между собой файлов. Подобно обычному приложению дал настольной системы, это делается для удобства разработчиков, поскольку иерархическая структура может сделать большое множество файлов более понятным.

Если у вас есть Web-приложение ASP.NET, содержащее необязательные подкаталога в корневом каталоге, вы с удивлением обнаружите, что *каждый*такой подкаталог может иметь собственный файл Web.config. Это дает возможность каждому подкаталогу переопределить установки родительского каталога. Если подкаталог не имеет своего пользовательского файла Web.config, каталог наследует установки следующего файла Web.config, размещенного выше по структуре каталога. Такой подход как это ни странно звучит, позволяет перенести определенные принципы ООП на структуру каталогов. Соответствующая концепция иллюстрируется схемой, представленной на рис. 24.12.

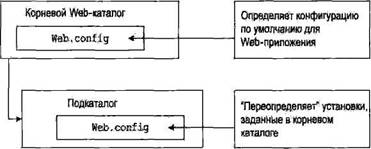


Рис. 24.12. Наследование конфигурации

Конечно, хотя ASP.NET позволяет определить множество файлов Web.config для одного Web-приложения, это делать совсем не обязательно. В большинстве случаев Web-приложению для нормального функционирования будет вполне достаточно одного файла Web.config, размещенного в корневом виртуальном каталоге IIS.

**Замечание**. Вспомните из плавы 11, что файл machine.config определяет различные установки на уровне машины, многие из которых связаны с ASP.NET. Этот файл является наивысшим в иерархии наследования конфигурации.

На этом наш обзор ASP.NET завершается. Как уже подчеркивалось в главе 23, полное и исчерпывающее рассмотрение ASP.NET 2.0 требует отдельной и довольно объемной книги. Но я надеюсь, что теперь вы чувствуете себя достаточно уверенно в рамках соответствующей программной модели.

**Замечание.**Если вам требуется более глубоко изучить ASP.NET 2.0, обратитесь к книге Мэтью Мак-Дональда и Марио Шпушта, *Microsoft ASP.NET 2.0 с примерами на*C# *2005 для профессионалов*(ИД "Вильямс", 2006 г.).

В завершение нашего путешествия в последней главе будет рассмотрена тема создания Web-сервисов XML в .NET 2.0.

**Резюме**

В этой главе вы имели возможность расширить свои знания в области ASP.NET, рассмотрев возможности использования типа HttpApplication. Вы могли убедиться в том, что этот тип предлагает целый ряд обработчиков событий уровня приложения и сеанса.

Значительная часть этой главы была посвящена рассмотрению различных подходов к управлению данными состояния. Напомним, что данные состояния представлений используются для автоматического обновления значений HTML-элементов при вторичных обращениях к Web-странице. Затем были выяснены различия между данными уровня приложения и сеанса, рассмотрены возможности управления данными cookie и изучен кэш приложения ASP.NET. Наконец, был рассмотрен набор элементов, которые могут присутствовать в файле Web.config.

**ГЛАВА 25. Web-сервисы XML**

Глава 18 информировала вас о слое удаленного взаимодействия .NET. Вы смогли убедиться в том, что эта технология позволяет любой группе компьютеров с поддержкой .NET осуществлять обмен информацией через границы машин. Это, конечно, прекрасно, но одним из ограничений слоя удаленного взаимодействия .NET оказывается то, что для каждой из участвующих в обмене сторон требуется установка .NET Framework, поддержка CTS и использование одинакового формата сетевого обмена (например, TCP).

Web-сервисы XML предлагают более гибкую альтернативу в деле построения распределенных приложений. Говоря простыми словами, *Web-cepвuc XML -*это единица программного кода, обслуживаемая Web-сервером и доступная в рамках стандартных промышленных технологий, например, таких как HTTP и XML. Вы, наверное, догадываетесь, что благодаря использованию "нейтральных" технологий Web-сервисы XML предлагают такой уровень совместимости и взаимодействия в отношении операционных систем, платформ и языков, который ранее был просто недоступен.

Из этой последней главы книга вы узнаете о том, как создаются Web-сервисы XML в рамках платформы .NET. В процессе обсуждения основной темы мы рассмотрим также ряд связанных вопросов, в частности службы поиска (UDDI и DISCO), язык WSDL и протокол SOAP. Выяснив, как строить Web-сервисы XML, мы рассмотрим различные возможности генерирования агентов клиента, способных вызывать "Web-методы" в синхронном и асинхронном режимах.

**Роль Web-сервисов XML**

С точки зрения самого высокого уровня вы можете определить Web-сервис XML, как единицу программного кода, доступную для вызова с помощью HTTP-запросов. Однако, в отличие от традиционного Web-приложения, Web-сервисы XML можно использовать не только для того, чтобы возвращать браузеру HTML-код с целью визуализации. Скорее наоборот, Web-сервис XML чаще всего предоставляет функциональные возможности, аналогичные возможностям стандартной библиотеки программного кода .NET (например, специальные вычисления, выборку данных из DataSet, чтение цен на акции и т.д.).

**Преимущества Web-сервисов XML**

На первый взгляд, Web-сервисы XML могут показаться просто очередной новой технологией удаленного взаимодействия. Это, конечно, так и есть, но давайте рассмотрим эту технологию чуть подробнее. Исторически для доступа к удаленным объектам всегда требовались специальные зависящие от платформы (a часто и от языка) протоколы (DCOM, Java RMI и т.д). Проблема такого подхода заключается не в лежащей в его основе технологии, а в том, что каждая из сторон замыкается в своем специфическом сетевом формате. Поэтому при попытке построения распределенной системы, в которой используется множество операционных систем, каждой машине приходится согласовывать формат пакета данных, протокол передачи и т.д. С целью упрощения ситуации Web-сервисы XML позволяют вызывать методы и свойства удаленного объекта с помощью стандартных HTTP-запросов. Из всех протоколов, существующих на сегодняшний день, HTTP является единственным сетевым протоколом, с которым "согласны" все платформы (в конце концов, HTTP – это основа World Wide Web).

Другой фундаментальной проблемой использования частных архитектур удаленного взаимодействия является то, что все они требуют, чтобы отправитель и получатель "понимали" одну и ту же систему базовых типов. Однако, и вы с этим должны согласиться, arrayList Java имеет мало общего с ArrayList .NET, и оба они не имеют ничего общего с массивом C++, Web-сервисы XML обеспечивают возможность гармоничного обмена информацией для несовместимых платформ, операционных систем и языков программирования. Вместо того чтобы вынуждать вызывающую сторону понимать специальную систему типов, информация между системами передается в виде XML-данных (которые на поверку оказываются "правильно" форматированными строками). Основным правилом здесь является следующее: если ваша операционная система позволяет оперативный доступ и анализ символьных данных, она способна взаимодействовать и с Web-сервисом XML.

Замечание. Web-сервис XML Microsoft .NET производственного уровня о6служивается сервером IIS в рамках отдельного виртуального каталога. Однако, как говорилось в главе 23, с помощью WebDev.WebServer.exe в .NET 2.0 теперь можно загружать Web-содержимое и из локального каталога (при разработке и тестировании).

**Определение клиента Web-сервиса XML**

Одной особенностью Web-сервисов XML, которая может сначала казаться непонятной, является то, что "потребителем" Web-сервисов XML являются не только Web-страницы. Консольные клиенты и клиенты Windows Forms тоже могут использовать Web-сервисы. В любом случае потребитель Web-cервиса XML неявно взаимодействует с удаленным Web-сервисом XML, с помощью промежуточного типа агента (proxy).

Агент Web-сервиса XML выглядит и ведет себя в точности так, как настоящий удаленный объект, предлагая при этом тот же набор членов. Однако "за кулисами" программный код агента направляет запросы Web-сервису XML, используя стандартные возможности HTTP. Агент также отображает поступающий поток XML-данных в соответствующие типы данных .NET (или любую другую систему типов, понятных приложению потребителя). На рис. 25.1 показана базовая схема взаимодействия Web-сервисов XML.

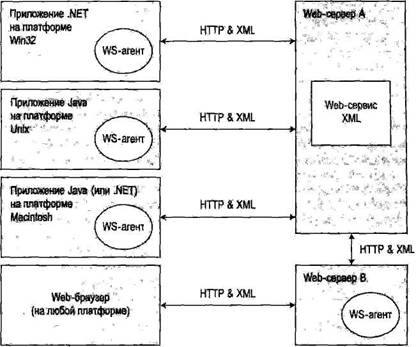


Рис. 25.1. Web-сервисы XML в действии

**Компоненты Web-сервиса XML**

В дополнение к библиотеке управляемого программного кода, обеспечивающей предлагаемые сервисом функциональные возможности, для Web-сервиса XML требуется определенная инфраструктура поддержки. В частности, Web-сервис XML использует следующие базовые технологии:

• служба поиска (позволяющая клиентам выяснить место нахождения Web-сервиса XML);

• служба описания (позволяющая клиентам узнать, что может предложить Web-сервис XML);

• транспортный протокол (позволяющий обмениваться информацией между клиентом и Web-сервисом XML).

Мы рассмотрим подробно каждый элемент инфраструктуры в процессе изучения материала этой главы. Но для начала обсуждения ниже предлагается краткий обзор указанных технологий поддержки.

**Служба поиска Web-сервиса XML**

Перед тем как клиент сможет использовать функциональные возможности Web-сервиса, ему нужно узнать о существовании и месте размещения этого сервиса. Если вы являетесь создателем и клиента, и Web-сервиса XML, фаза поиска оказывается очень простой, поскольку вы сами являетесь источником нужной информации. Но что делать, если вы хотите сделать возможности вашего Web-сервиса открытыми для всех?

Для этого вы можете зарегистрировать свой Web-сервис XML на сервере UDDI (Universal Description, Discovery, and Integration – универсальное описание, поиск и взаимодействие). Клиенты могут послать запрос к каталогу UDDI, чтобы получить список всех Web-сервисов, соответствующих заданным критериям поиска (например, "найти все Web-сервисы, связанные с получением метеорологических данных в реальном времени"). Идентифицировав подходящий Web-сервер в списке, возвращенном в результате UDDI-запроса, вы можете выяснить все возможности этого сервера. Если хотите, можете назвать UDDI "белой книгой" Web-сервисов XML.

В дополнение к UDDI-поиску, Web-сервис XML, построенный в рамках .NET, можно найти с помощью DISCO – этот несколько искусственный акроним расшифровывается, как *Discovery of Web Services*(поиск Web-сервисов). Используя файл статического поиска (\*.disco) или динамического поиска (\*.vsdisco), вы можете "афишировать" набор Web-сервисов XML, размещенных по конкретному адресу URL. Потенциальные клиенты Web-сервисов могут перейти к файлу \*.disco Web-сервера, чтобы проверить связи всех опубликованных Web-сервисов XML.

Следует учитывать то, что по умолчанию динамический поиск отключен, поскольку имеется потенциальный риск нарушения защиты, если позволить IIS открыть весь набор Web-сервисов XML всем интересующимся объектам. В связи с этим службы DISCO здесь обсуждаться не будут.

**Замечание**. Если вы захотите активизировать поддержку динамического поиска для Web-сервера, прочитайте статью Q307303 базы знаний Microsoft на страницах [http://support.microsoft.com](http://support.microsoft.com/)

**Служба описания Web-сервиса XML**

Итак, клиент знает, где размещен Web-сервис XML. Теперь клиент должен узнать функциональные возможности этого сервиса. Например, клиент должен иметь возможность узнать, что сервис имеет метод GetWeatherReport(), предполагающий использование некоторого набора параметров и возвращающий некоторое значение, чтобы клиент мог вызвать этот метод. Вы, возможно, догадываетесь, что это предполагает использование некоторого метаязыка, нейтрального в отношении всех платформ, языков и операционных систем. XML-метаданные, используемые для описания Web-сервисов XML, создаются на *языке WSDL*(Web Services Description Language – язык описания Web-сервисов).

Во многих случаях WSDL-описание Web-сервиса XML автоматически генерируется сервером IIS Microsoft, если поступающий запрос имеет суффикс ?wsdl. Вы увидите, что первичными потребителями WSDL-контрактов являются инструменты генерирования агентов. Например, утилита командной строки wsdl.exe (ее обсуждение будет предложено позже) генерирует клиентский C#-класс агента на основе имеющегося WSDL-документа.

В более сложных случаях (обычно с целью гарантии совместимости) при построении Web-сервисов многие разработчики используют подход, в рамках которого сначала вручную определяется WSDL-документ, поскольку упомянутая выше утилита командной строки wsdl.exe может генерировать описания интерфейса для Web-сервиса XML и на основе WSDL-определения.

**Транспортный протокол**

После создания типа агента для взаимодействия с Web-сервисом XML клиент может вызывать доступные методы. Как уже подчеркивалось, соответствующие данные передаются с помощью сетевого протокола HTTP. В частности, для обмена информацией между потребителями и Web-сервисами можно использовать HTTP-методы GET и POST или SOAP.

В общем, основным вариантом выбора обычно оказывается SOAP, поскольку, как вы вскоре убедитесь, сообщения SOAP могут содержать XML-описания сложных типов (включая пользовательские типы и типы из библиотек базовых классов .NET), При использовании HTTP-протоколов GET и POST вам придется ограничиться более узким множеством типов XML-схемы.

**Пространства имен .NET для Web-сервисов XML**

Теперь, когда у вас есть база для понимания принципов функционирования Web-сервисов XML, мы с вами можем заняться построением такого объекта в рамках платформы .NET. Библиотеки базовых классов определяют целый ряд пространств имен, обеспечивающих взаимодействие с любой из доступных технологий использования Web-сервисов (табл. 25.1).

Таблица 25.1. Пространства имен для работы с Web-сервисами XML

|  |  |
| --- | --- |
| **Пространство имен** | **Описание** |
| System.Web.Services | Содержит базовые типы (включая очень важный атрибут [WebMethod]), необходимые для построения любого Web-сервиса XML |
| System.Web.Services.Configuration | Содержит типы, позволяющие настроить поведение Web-сервиса XML в среде выполнения ASP.NET |
| System.Web.Services.Description | Содержит типы, обеспечивающие программное взаимодействие с WSDL-документом, предлагающим описание данного Web-сервиса |
| System.Web.Services.Discovery | Содержит типы, позволяющие потребителям Web-сервисов выполнять программный поиск Web-сервисов на соответствующей машине |
| System.Web.Services.Protocols | Определяет ряд типов, представляющих "атомы" различных протоколов связи Web-сервисов XML (HTTP-методы get и POST, а также SOAP) |

**Замечание.**Все пространства имен, связанные с Web-сервисами XML, содержатся в компоновочном блоке System.Web.Services.dll.

**Пространство имен System.Web.Services**

Несмотря на богатые функциональные возможности, обеспечиваемые всеми пространствами имен .NET, связанными с Web-сервисами XML, подавляющее большинство ваших приложений потребует непосредственного взаимодействия только с типами, определенными в System.Web.Services. Как становится ясно из табл. 25.2, количество таких типов достаточно невелико (что уже хорошо).

**Таблица 25.2.** Члены пространства имен System.Web.Services

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| WebMethodAttribute | Добавление атрибута [WebMethod] в метод или свойство типа класса Web-сервиса обозначает возможность вызова соответствующего члена средствами HTTP и сериализацию в формате XML |
| WebService | Опциональный базовый класс построения Web-сервисов XML в .NET. При использовании этого класса производный Web-сервис XML будет иметь возможность "аккумулировать" информацию состояния (например, переменные сеанса и приложения) |
| WebServiceAttribute | Атрибут [WebService] может использоваться для добавления в Web-сервис информации, например, такой как строка с описанием функциональных возможностей сервиса и соответствующих пространств имен XML |
| WebServiceBindingAttribute | Этот атрибут (появившийся в .NET 2.0) объявляет связывающий протокол, реализуемый данным методом Web-сервиса (HTTP-протоколы get и POST или SOAP), и уровень функциональной совместимости (WSI) Web-сервиса |
| WsiProfiles | Этот перечень (появившийся в .NET 2.0) используется для описания спецификаций WSI (Web Services Interoperability – функциональная совместимость Web-сервисов), которым должен удовлетворять данный Web-сервис |

Остальные пространства имен, показанные в табл. 25.1, могут быть полезны вам только в том случае, если вы захотите вручную взаимодействовать с WSDL-документом, службами поиска или соответствующими сетевыми протоколами. Все подробности можно найти в документации .NET Framework 2.0 SDK.

**Создание Web-сервиса XML вручную**

Как и любое другое приложение .NET, Web-сервисы XML можно создавать вручную, без использования интегрированной среды разработки, такой как, например, Visual Studio 2005. Чтобы прояснить возможности использования Web-сервисов XML, давайте построим пример простого Web-сервиса XML вручную. С помощью текстового редактора создайте новый файл с именем HelloWorldWebService.asmx (по умолчанию для обозначения файлов Web-сервисов .NET используется расширение \*.asmx). Сохраните файл в подходящем месте на своем жестком диске (например, в папке C:\HelloWorldWebService), добавив следующее определение типа.

**‹%@ WebService Language="C#" Class="HelloWebService.HelloService"**%›

using System;

using System.Web.Services;

namespace HelloWebService {

 public class HelloService {

**[WebMethod]**

  public string HelloWorld() {

   return "Hello!";

  }

 }

}

В основном, файл \*.asmx выглядит аналогично любому другому определению пространства имён C#. Первым достойным внимания отличием является то, что здесь используется директива ‹%@WebService%›, которая должна, как минимум, указать название управляемого языка, используемого для определения соответствующего класса, и полное имя этого класса. В дополнение к атрибутам Language и Class директива ‹%@WebService%› может также содержать атрибут Debug, информирующий компилятор ASP.NET о необходимости генерирования символов отладки, и необязательное значение CodeBehind, идентифицирующее связанный файл программного кода поддержки в пределах необязательного каталога App\_Code (см. главу 23). В этом примере мы не собираемся использовать внешний файл кода поддержки, а встроим всю необходимую программную логику непосредственно в файл

Кроме использования директивы ‹%@WebService%›, другой особенностью это-го файла \*.asmx является использование атрибута [WebMethod], информирующего среду выполнения ASP.NET о том, что этот метод будет доступен для поступающих HTTP-запросов и должен позволять сериализацию возвращаемых значений в формате XML.

**Замечание**. В рамках HTTP могут быть доступными только члены, имеющие атрибут [WebMethod]. Члены, не обозначенные атрибутом [WebMethod], не могут вызываться агентом клиента

**Тестирование Web-сервиса XML с помощью WebDev.WebServer.exe**

Напомним (снова см. главу 23), что WebDev.WebServer.exe является сервером Web-разработки ASP.NET, поставляемым в составе дистрибутива .NET Framework 2.0 SDK. И хотя WebDev.WebServer.exe не предполагается использовать для обслуживания Web-сервисов XML производственного уровня, этот инструмент позволяет запустить Web-содержимое непосредственно из локального каталога при отладке. Для проверки своего сервиса с помощью этого инструмента откройте окно командной строки Visual Studio 2005 и выполните следующую команду, указав свободный номер порта и физический путь к каталогу, содержащему ваш файл \*.asmx.

WebDev.Webserver /port:4000 /path:"C:\HelloWorldWebService"

После запуска Web-сервера откройте любой браузер и укажите в его окне имя своего файла \*.asmx, используя соответствующий номер порта.

http://localhost:4000/HelloWorldWebService.asmx

Вам будет показан список всех Web-методов, доступных по этому адресу URL (рис. 25.2).



Рис. 25.2. Тестирование Web-сервиса XML

Если в окне браузера вы щелкнете на ссылке HelloWorld, откроется другая страница, которая позволит вызвать [WebMethod], только что выбранный вами. В результате вызова HelloWorld() будет возвращена не буквальная строка .NET System.String, a XML-представление текстовых данных, возвращаемых Web-методом HelloWorld().

‹?xml version="1.0" encoding="utf-8"?›

‹string xmlns="bttp://tempuri.org/**"›Hello!**‹/string›

**Тестирование Web-сервиса XML с помощью IIS**

Теперь, когда вы проверили свой Web-сервис XML с помощью WebDev. WebServer.exe, перенесите файл \*.asmx в виртуальный каталог IIS. Используя инструкции, предложенные в главе 23, создайте новый виртуальный каталог с именем HelloWS, который будет отображаться в физическую папку, содержащую файл HelloWorldWebServiсe.asmx. После этого вы получите возможность проверить свой Web-сервис с помощью ввода следующего значения URL в строке Web-браузера.

http://localhost/HelloWS/HelloWorldWebService.asmx

**Просмотр WSDL-документа**

Как уже упоминалось, WSDL является метаязыком, описывающим многочисленные особенности Web-методов, доступных по данному адресу URL. Обратите внимание на то, что при проверке Web-сервиса XML автоматически генерируемая страница тестирования предлагает ссылку Service Description (Описание сервиса). В результате щелчка на этой ссылке к текущему запросу присоединяются символы ?wsdl. Когда среда выполнения ASP.NET получает запрос для файла \*.asmx с таким прикрепленным суффиксом, она автоматически возвращает соответствующий WSDL-код, открывающий каждый доступный Web-метод.

В настоящий момент вам не следует беспокоиться о природе WSDL-кода или формате WSDL-документа. Пока что важно только понимать, что WSDL-код описывает то, как Web-методы могут вызываться с помощью имеющихся протоколов связи Web-сервиса XML.

**Автоматически генерируемая страница тестирования**

Как вы только что убедились, работоспособность Web-сервисов XML можно проверить с помощью автоматически генерируемой HTML-страницы в браузере. Когда обнаруживается HTTP-запрос, указывающий на данный файл \*.asmx, среда выполнения ASP.NET использует файл с именем DefaultWsdlHelpGenerator.aspx, чтобы создать HTML-страницу, позволяющую вызвать Web-методы, доступные по данному URL. Этот файл \*.aspx можно найти в следующем каталоге (здесь, конечно, блок ‹версия› следует заменить на номер вашей текущей версии .NET Framework).

C:\Windows\Microsoft.NET\Framework\‹версия›\CONFIG

**Создание пользовательской страницы тестирования**

Если вы хотите, чтобы среда выполнения ASP.NET применяла пользовательский файл \*.aspx для проверки ваших Web-сервисов XML, вы можете встроить в эту страницу дополнительную информацию (например, фирменный знак компании, дополнительные описания сервиса, ссылки на файлы справки и т.д.). Чтобы упростить себе задачу, большинство разработчиков сначала копируют существующий файл DefaultWsdlHelpGenerator.aspx в проект, а затем, используя этот файл в качестве исходного, нужным образом изменяют оригинальный HTML-документ и программный код C#.

Скопируйте файл DefaultWsdlHelpGenerator.aspx в каталог, содержащий HelloWorldWebService.asmx (например, C:\HelloWorldWebService). Переименуйте полученную копию в MyCustomWsdlHelpGenerator.aspx и измените какой-нибудь фрагмент HTML-кода, скажем, в области дескриптора ‹title›. Например, измените имеющийся код разметки

‹title›‹%#ServiceName + " " + GetLocalizedText("WebService'')%›‹/title›

на следующий.

‹title›**Мой собственный**

 ‹%#ServiceName + " " + GetLocalizedText("WebService") %›

‹/title›

После изменения HTML-содержимого создайте файл Web.config и сохраните его в текущем каталоге. Следующие XML-элементы дают указание среде выполнения использовать ваш пользовательский файл \*.aspx, а не DefaultWsdlHelpGenerator.aspx.

**‹!--Здесь указывается пользовательский файл \*.aspx --›**

‹configuration›

 ‹system.web›

  ‹webServices›

   ‹wsdlHelpGenerator href="**MyCustomWsdlHelpGenerator.aspx**" /›

  ‹/webServices›

 ‹/system.web›

‹/configuration›

При запросе своего Web-сервиса вы увидите, что строка заголовка браузера изменится в соответствии с указанным вами пользовательским содержимым. Кстати, если вы захотите отключить генерирование страницы помощи для данного Web-сервиса, вы можете сделать это с помощью элемента ‹remove› в файле Web.config.

**‹!-- Отмена генерирования страницы помощи --›**

‹configuration›

 ‹system.web›

  ‹webServices›

   ‹protocols›

**‹!-- Этот элемент отменяет генерирование WSDL-документа --›**

    ‹remove name="Documentation"/›

   ‹/protocols›

  ‹/webServices›

 ‹/system.web›

‹/configuration›

**Исходный код.**Файлы примера HelloWorldWebService размещены в подкаталоге, соответствующем главе 25.

**Создание Web-сервиса XML в Visual Studio 2005**

Создав Web-сервис XML вручную, давайте посмотрим, как это делается в Visual Studio 2005. Выбрав File→New→Web Site из меню, создайте новый C#-проект Web-сервиса XML с именем MagicEightBallWebService и сохраните этот проект на своем локальном диске (рис. 25.3).

**Замечание.**Как и в случае Web-узла ASP.NET, файлы \*.sln проектов Web-сервисов XML, созданных в Visual Studio 2005, размещаются в папке Мои документы\Visual Studio 2005\ Projects.

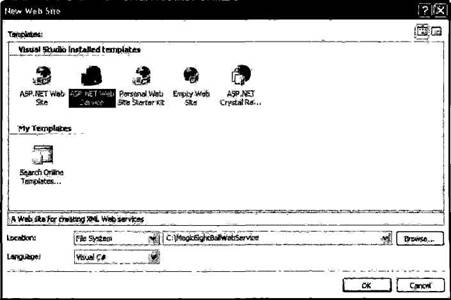


Рис. 25.3. Проект Web-сервиса XML в Visual Studio 2005

После щелчка на кнопке OK в окне создания сервиса Visual Studio 2005 будет сгенерирован файл Service.asmx, определяющий следующую директиву ‹%@WebService%›.

‹%@ WebService Language="C#" CodeBehind="~/App\_Code/Servicе.cs" Class="Service" %›

Обратите внимание на то, что здесь используется атрибут CodeBehind, чтобы указать имя файла с программным кодом C#, определяющим соответствующий тип класса (этот файл по умолчанию размещается в каталоге App\_Code проекта). По умолчанию Service.cs определяется так.

using System;

using System.Web;

using System.Web.Services;

using System.Web.Services.Protocols;

[WebService(Namespace="http://tempuri.org/")]

[WebServiceBinding(ConformsTo=WsiProfiles.BasicProfile1\_1)]

public class Service: System.Web.Services.WebService {

 public Service() { }

 public string HelloWorld() {

  return "Hello World";

 }

}

В отличие от предыдущего примера HelloWorldWebService, здесь класс Service получается из базового класса System.Web.Services.WebService. Члены, определенные этим типом, будут рассмотрены чуть позже, а здесь достаточно подчеркнуть, что получать класс Service именно из этого базового класса совсем не обязательно.

Также обратите внимание на то, что класс Service имеет два (также необязательных) атрибута, [WebService] и [WebServiceBinding]. Роль этих атрибутов тоже будет рассмотрена немного позже.

**Реализация Web-метода TellFortune()**

Ваш Web-сервис XML MagicEightBall будет имитировать классическую говорящую игрушку, сообщающую предсказания. Для этого добавьте в класс Service следующий метод (существующий Web-метод HelloWorld() можно удалить).

[WebMethod]

public string TellFortune(string вопросПользователя) {

 string[] answers *=*{"Будущее неоднозначно", "Да", "Нет", "Вряд ли", "Спросите еще раз", "Определенно" };

**// Возвращение случайного ответа на вопрос.**

 Random r = new Random();

 return string.Format("{0}? {1}", вопросПользователя, answers[r.Next(answers.Length)]);

}

Для проверки нового Web-сервиса XML просто запустите проект на выполнение (или для отладки) в Visual Studio 2005. Поскольку для метода ТеllFortune() требуется один входной параметр, автоматически генерируемая HTML-страница тестирования обеспечивает необходимое поле ввода (рис. 25.4).

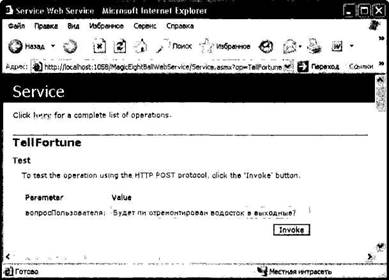


Рис. 25.4. Вызов Web-метода TellFortune()

Вот возможный ответ на вопрос "Будет ли отремонтирован водосток в выходные?"

‹?xml version="1.0" encoding="utf-8"?›

‹string xmlns="http://tempuri.org/"›"Будет ли отремонтирован водосток в выходные? Вряд ли

*‹/*string*›*

Итак, к этому моменту вы создали два простых Web-сервиса XML: один вручную, а другой – с помощью Visual Studio 2005. Теперь у вас есть хорошая основа для углубленного обсуждения соответствующих вопросов, и начнем мы это обсуждение с рассмотрения роли базового класса

**Исходный код.**Файлы примера MagicEightBallWebService размещены в подкаталоге, соответствующем главе 25.

**Роль базового класса WebService**

В процессе разработки сервиса HelloWorldWebService вы имели возможность убедиться том, что Web-сервис можно получить непосредственно из System.Object. Но по умолчанию Web-сервисы, созданные в Visual Studio 2005, автоматически получаются из базового класса System.Web.Services.WebService. Описания основных членов этого типа класса предлагаются в табл. 25.3.

Таблица 25.3. Основные члены типа System.Web.Services.WebService

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Application | Обеспечивает доступ к объекту HttpApplicationState для текущего HTTP-запроса |
| Context | Обеспечивает доступ к типу HttpContext, инкапсулирующему все НТТР-содержимое, используемое Web-сервером для HTTP-запросов |
| Server | Обеспечивает доступ к объекту HttpServerUtility для текущего запроса |
| Session | Обеспечивает доступ к типу HttpSessionState для текущего запроса |
| SoapVersion | Читает версию протокола SOAP, используемую для SOAP-запросов к Web-сервису XML: это свойство появилось в .NET 2.0 |

Вы, возможно, уже поняли, что для построения Web-сервиса, способного осуществлять поддержку своего состояния с помощью переменных приложения и сеанса (см. главу 24), вы должны получить соответствующий тип из WebService, поскольку последний определяет свойства Aррlication и Session. С другой стороны, если вы строите Web-сервис XML, для которого не требуется "помнить" информацию о внешних пользователях, не требуется и расширение WebService. Мы снова рассмотрим процесс построения Web-сервиса XML позже, в ходе нашего обсуждения свойства EnableSession атрибута [WebMethod].

**Атрибут [WebService]**

Класс Web-сервиса XML может быть помечен необязательным атрибутом [WebService] (не путайте его с базовым классом WebService). Этот атрибут поддерживает ряд именованных свойств, первым из которых является Namespace. Это свойство можно использовать для указания пространства имен XML, используемого в документе WSDL.

Вы, возможно, уже знаете, что пространства имен XML используются для создания контекста применения пользовательских XML-элементов в рамках конкретной группы (точно так же, как и пространства имен .NET). По умолчанию среда выполнения ASP.NET назначает для файла \*.asmx фиктивное пространство имен XML http://tempuri.org. Аналогично по умолчанию Visual Studio 2005 назначает для Namespace значение http://tempuri.org.

Предположим, что в Visual Studio 2005 вы создали новый проект Web-сервиса XML с именем CalculatorService, определяющий следующие два Web-метода с именами Add() и Subtract().

[WebService(Namespace = "http://tempuri.org/")]

[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]

public class Service: System.Web.Services.WebService {

 [WebMethod]

 public int Subtract (int x, int y) { return x – y; }

 [WebMethod]

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

}

Перед публикацией своего Web-сервиса XML вы должны указать соответствующее пространство имен, которое обычно представляет собой адрес URL узла, обслуживающего Web-сервис. В следующем варианте программного кода обратите внимание на то, что атрибут [WebService] позволяет установить именованное свойство Description, предлагающее описание вашего Web-сервиса.

[WebService**(Description**= "Чудесный Web-сервис калькулятора",

**Namespace**="http://www.IntertechTraining.com/")]

[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]

public class Service: System.Web.Services.WebService {…}

**Свойства Namespace и Description**

При запуске этого проекта вы обнаружите, что теперь автоматически сгенерированная страница тестирования не отображает сообщение с предложением заменить http://tempuri.org. Более того, если вы щелкнете на ссылке Service Description, чтобы просмотреть содержимое соответствующего документа WSDL, вы увидите, что атрибут TargetNamespace соответствует указанному вами пользовательскому пространству имен XML. Наконец, WSDL-файл теперь содержит элемент ‹documentation›, соответствующий указанному вами значению Description.

‹wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"›

**Чудесный Web-сервис калькулятора**

‹/wsdl:documentation›

Вы, наверное, уже догадались что вполне возможно построить пользовательскую утилиту (например, объектный браузер Web-сервиса XML) для считывания значений, содержащихся в контексте элемента ‹documentation›. Однако в большинстве случаев соответствующее значение будет использоваться файлом DefaultWsdlHelpGenerator.aspx.

**Свойство Name**

Последним из рассматриваемых здесь свойств типа WebServiceAttribute является свойство Name, которое используется для указания имени Web-сервиса XML, водимого внешним пользователем. По умолчанию внешнее имя Web-сервиса идентично имени соответствующего типа класса (которым, в свою очередь по умолчанию является имя Service). Однако если вы хотите "отделить" имя класса .NET от соответствующего WSDL-имени, вы можете изменить атрибут [WebService] так, как показано ниже.

[WebService(**Description** = "Чудесный Web-сервис калькулятора",

**Namespace**= "http://www.IntertechTraining.com/",

**Name**= "CalculatorWebService")];

[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]

{…}

На рис. 25.5 показана страница тестирования, автоматически сгенерированная с помощью DefaultWsdlHelpGenerator.aspx с учетом указанного значения атрибута [WebService].

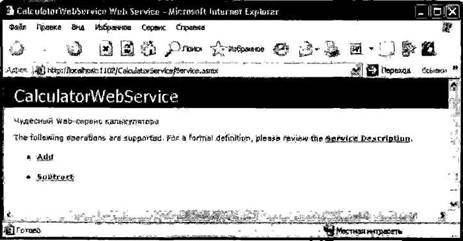


Рис 25.5. Web-сервис CalculatorWebService

**Атрибут [WebServiceBinding]**

В .NET 2.0 Web-сервис XML может содержать атрибут [WebServiceBinding]. Среди прочего этот атрибут используется для того, чтобы указать соответствие данного Web-сервиса XML "базовому профилю совместимости Web-сервисов (WSI) версии 1.1". Но что это значит? Если вы активно работаете с Web-сервисами XML, вы должны знать, что спецификации WSDL в процессе развития этой технологии изменялись. Вследствие этого вполне обычной оказывается ситуация, когда один и тот же элемент (или атрибут) WSDL имеет разные интерпретации в разных системах разработки (IIS, WSAD), Web-серверах (IIS, Apache) и архитектурах (.NET, J2EE).

Очевидно, для Web-сервиса XML это оказывается проблемой, поскольку одной из задач разработки является упрощение способа обработки информации в межплатформенном мультиархитектурном и многоязычном окружении. Чтобы решить проблему, для повышения уровня межплатформенной совместимости Web-сервисов организация WSI предлагает использовать открытые спецификации. В .NET 2.0 свойству ConformsTo атрибута [WebServiceBinding] можно назначить любое значение из перечня WsiProfiles.

public enum WsiProfiles {

**// Web-сервис не делает никаких заявлений о совместимости.**

 None,

**// Web-сервис объявляет о совместимости**

**// с базовым профилем WSI версии 1.1.**

 BasicProfile1\_1

}

По умолчанию Web-сервисы XML, генерируемые с помощью Visual Studio 2005, предполагают согласованность с базовым профилем WSI 1.1. Конечно, простое приравнивание свойства ConformsTo к WsiProfiles.BasicProfile1\_1 не гарантирует, что каждый Web-метод будет действительно совместим с этим профилем. Например, одно из правил ВР 1.1 гласит, что каждый метод WSDL-документа должен иметь свое уникальное имя (т.е. ВР 1.1 не допускает перегрузку Web-методов). Но хорошей вестью здесь является то, что среда выполнения ASP.NET позволяет определить различные уровни соответствия ВР 1.1 и сообщит об этом во время выполнения.

**Игнорирование проверки соответствия правилам ВР 1.1**

В .NET 2.0 Web-сервисы XML автоматически проверяются на соответствие спецификациям базового профиля WSI версии 1.1 (ВР 1.1). В большинстве случаев это хорошо, поскольку позволяет создавать программное обеспечение с самыми широкими возможностями совместимости. Однако в некоторых случаях бывает нужно пренебречь совместимостью с ВР 1.1 (например, при создании внутренних Web-сервисов XML, когда совместимость не столь важна). Чтобы дать указание среде выполнения игнорировать нарушение правил ВР 1.1, установите свойство ConformsTo равным WsiProfiles.None, а свойство EmitConformanceClaims – равным false (ложь).

[WebService(Description = "Чудесный Web-сервис калькулятора",

 Namespace = "http://www.IntertechTraining.com/",

 Name = "CalculatorWebService")]

[WebServiceBinding(**ConformsTo =**WsiProfiles.None,

**EmitConformanceClaims***=*false")]

public class Service: System.Web.Services.WebService {…}

Как и следует ожидать, от значения, присвоенного свойству EmitConformanceClaims, зависит, будут ли при публикации WSDL-описания Web-сервиса учиты-ваться сообщения соответствия свойства ConformsTo. В данном случае нарушение правил ВР 1.1 разрешается, но автоматически генерируется страница тестирования все равно будет отображать предупреждения.

**Отмена проверки соответствия правилам BP 1.1**

Чтобы полностью отключить проверку соответствия BP 1.1 для Web-сервиса XML, определите в соответствующем файле Web.соnfig элемент

‹conformanceWarnings›.

 ‹configuration›

  ‹webServices›

   ‹conformanceWarnings›

**‹remove name="BasicProfile1\_1"**/›

   ‹/conformanceWarnings›

  ‹/webServices›

 ‹/system.web›

‹/configuration›

**Замечание**. Атрибут [WebServiceBinding] можно также использовать для определения предполагаемых связей конкретных методов по значению свойства Name. Соответствующие подробности можно найти в документации .NET Framework 2.0 SDK.

**Атрибут [WebMethod]**

Атрибут [WebMethod] должен указываться для каждого метода, который вы хотите сделать доступным в рамках данного Web-сервиса XML. Как и большинство других атрибутов, тип WebMethodAttiibute может иметь целый ряд необязательных именованных свойств. Давайте рассмотрим каждую из имеющихся здесь возможностей по очереди.

**Описание Web-метода с помощью свойства Description**

Как и в случае атрибута [WebService], свойство Description атрибута [WebMethod] позволяет описать функциональные возможности Web-мeтoдa.

public class Service: System.Web.Services.WebService {

 [WebMethod**(Description**= "Вычитание целых чисел.")]

 public int Subtract (int x, int y) { return x – y; }

 [WebMethod(Description = "Сложение целых чисел.")]

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

}

При указании свойства Description в пределах атрибута [WebMethod] в WSDL-документ в контексте имени метода добавляется новый элемент ‹documentation›.

‹wsdl:operation name="**Add**"›

 ‹wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"› **Выполняет сложение целых чисел.**

 ‹/wsdl:documentation›

 ‹wsdl:input message="tns:AddSoapIn" /›

 ‹wsdl:output message="tns:AddSoapOut" /›

‹/wsdl:operation›

**Устранение конфликтов имен WSDL с помощью свойства MessageName**

Одним из WSI-правил ВР 1.1 является то, что каждый метод в WSDL-документе должен быть уникальным. Таким образом, если вы хотите, чтобы ваш Web-сервис XML удовлетворял спецификациям ВР 1.1, вы не можете использовать перегруженные методы. Однако для примера предположим, что вы все-таки использовали перегрузку метода Add(), чтобы вызывающая сторона могла передать как два целых числа, так и два числа с плавающей точкой. В результате вы должны увидеть следующее сообщение среды выполнения.

Both Single Add(Single, Single) and Int32 Add(Int32, Int32) use the message name 'Add'. Use the MessageName property attribute to specify unique of the WebMethod custom message names for the methods.

(Сообщение гласит: Single Add(Single, Single) и Int32 Add(Int32, Int32) используют одно и то же имя 'Add' для своих сообщений; используйте свойство MessageName атрибута WebMethod, чтобы указать для этих методов уникальные пользовательские имена сообщений.)

Здесь лучшим решением является отказ от перегрузки метода Add(). Если же перегрузка необходима, для устранения конфликтов имен в документах WSDL можно использовать свойство MessageName атрибута [WebMethod].

public class Service: System.Web.Services.WebService {

 [WebMethod(Description = "Сложение чисел с плавающей точкой.",

**MessageName**= "AddFloats")]

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

 [WebMethod(Description = "Сложение целых чисел.",

**MessageName**= "AddInts")]

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

}

Поcле этого генерируемый WSDL-документ будет внутренне ссылаться на каждую из перегруженных версий метода Add() по уникальным именам (AddFloats и AddInts). Но с точки зрения агента на стороне клиента будет существовать только один перегруженный метод Add().

**Поддержка данных состояния Web-сервисов с помощью свойства EnableSession**

Вы, наверное, помните из главы 24 о том, что свойства Application и Session позволяют Web-приложению ASP.NET поддерживать данные состояния. Web-сервисы XML обеспечивают те же возможности с помощью базового класса System.Web.Services.WebService. Например, предположим, что ваш Web-сервис калькулятора поддерживает переменную уровня приложения (которая, таким образом, должна быть доступной любому сеансу), содержащую значение PI, как показано ниже.

public class CalcWebServicе: System.Web.Services.WebService {

**// Этот Web-метод обеспечивает доступ к переменной SimplePI**

**// уровня приложения.**

 [WebMethod(Description = "Получение значения РI.")]

 public float GetSimplePI() { return (float)**Application["SimplePI"];**}

 …

}

Начальное значение переменной SimplePI уровня приложения можно установить в обработчике Application\_Start(), определенном в файле Global.asax. Добавьте в свой проект глобальный класс приложения (щелкнув правой кнопкой мыши на пиктограмме проекта в окне обозревателя решений и выбрав Add New Item из появившегося меню), а затем измените Application\_Start() так, как предлагается ниже.

‹%@ Application Language="C#" %›

‹script runat="server"›

void Application\_Start(Object sender, EventArgs e) {

 Application["SimplePI"] =3.14F;

}

…

‹/script›

Вдобавок к поддержке переменных уровня приложения можно использовать свойство Session для поддержки сеансовой информации. Для примера реализуйте метод Session\_Start() в файле Global.asax так, чтобы каждый зарегистрированный пользователь идентифицировался случайным значением.

‹%@ Application Language="C#" %›

‹script runat= "server"›

…

void Session\_Start(Object sender, EventArgs e) {

**// Чтобы сделать доступными сеансовые данные Web-сервиса,**

**// присвойте каждому пользователю случайное число.**

 Random r = new Random ();

 Session["SessionRandomNumber"] = r.Next(1000);

}

…

‹/script›

С целью проверки в рамках класса Service создайте новый Web-метод, возвращающий присвоенное пользователю случайное значение.

public class Service: System.Web.Services.WebService {

 …

 [WebMethod**(EnableSession**= true,

  Description = "Получите ваше случайное значение!")]

 public int GetMyRandomNumber() { return (int)Session["SessionRandomNumber"]; }

}

Заметим, что здесь атрибут [WebMethod] явно устанавливает для свойства EnableSession значение true (истина). Этот шаг необходим, поскольку по умолчанию для любого Web-метода контроль его данных сеансового состояния *отключен.*Если вы теперь запустите два или три экземпляра браузера (чтобы сгенерировать множество идентификаторов сеанса), вы обнаружите, что каждый зарегистрировавшийся пользователь возвращает уникальное числовое значение. Например, первый пользователь может получить следующий XML-код:

‹?xml version="1.0" encoding="utf-8"?›

‹int xmlns="http://www.IntertechTraining.com/WebServers"›**931**‹/int›

в то время как второй может получить значение 472.

‹?xml verslon="l.0" encoding="utf-8"?›

‹int xmlns="http://www.IntertechTraining.com/WebServers"›**472**‹/int›

**Настройка данных сеансового состояния с помощью Web.config**

Наконец, напомним, что файл Web.config можно изменить с тем, чтобы в нем указывалось место, где должны запоминаться данные состояния для Web-сервиса XML. Для этого используют элемент ‹sessionState› (описанный в предыдущей главе).

‹sessionState mode="InProc" stateConnectionString="tcpip=127.0.0.1:42424" sqlConnectionString="data source=127.0.0.1;Trusted\_Connection=yes" cookieless="false" timeout="20" /›

**Исходный код.**Файлы примера CalculatorService размещены в подкаталоге, соответствующем главе 25.

**Язык описания Web-сервисов (WSDL)**

В последних нескольких примерах вы могли видеть отдельные фрагменты WSDL-кода. Напомним, что WSDL – это основанная на XML грамматика, предназначенная для описания возможностей взаимодействия внешних клиентов с Web-методами, доступными по данному адресу URL в рамках каждого из поддерживаемых протоколов связи. Во многих отношениях WSDL-документ может рассматриваться, как "контракт" между клиентом Web-сервиса и самим Web-сервисом. Это еще один метаязык. В частности, WSDL используется для описания следующих характеристик любого доступного Web-метода:

• имя Web-метода XML;

•  число, тип и порядок следования параметров (если таковые имеются);

• тип возвращаемого значения (если таковое предусмотрено);

• условия вызова HTTP GET, HTTP POST и SOAP.

В большинстве случаев WSDL-документы генерируются автоматически соответствующим Web-сервером. Напомним, что при добавлении суффикса?wsdl к адресу URL, указывающему на файл \*.asmx, Web-сервер генерирует WSDL-документ для указанного Web-сервиса XML.

http://locаlhost/SomeWS/theWS.asmx?wsdl

Но если IIS автоматически генерирует WSDL-документ для данного Web-сервиса XML, зачем тогда нужно глубокое понимание синтаксиса генерируемых WSDL-данных? Ответ обычно зависит от того, как ваш сервис будет использоваться внешними приложениями. В случае Web-сервисов XML, предназначенных для "внутреннего" использования, сгенерированного Web-сервером WSDL-кода будет, как правило, достаточно.

Между тем. вполне возможно начать разработку Web-сервиса XML с создания WSDL-документа вручную (об этом уже говорилось выше). Главная идея начала разработки с создания WSDL-документа связана с вопросами совместимости. Вспомните о том, что до появления спецификации WSI различные инструменты построения Web-сервисов нередко генерировали несовместимые WSDL-описания. Если начинать разработку с WSDL-кода, вы можете построить документ так, как требуется.

Как вы можете догадаться, для начала разработки Web-сервиса XML с создания WSDL-документа требуется очень хорошее знание грамматики WSDL, обсуждение которой в контексте этой главы не предусмотрено. Но мы рассмотрим базовую структуру WSDL-документа. Разобравшись в основах, вы сможете оценить пользу утилиты командной строки wsdl.exe.

**Замечание.**Самую свежую информацию о языке WSDL можно найти на страницах <http://www.w3.org/tr/wsdl>.

**Определение WSDL-документа**

Действительный документ WSDL открывается и закрывается корневым элементом ‹definitions›. В открывающем дескрипторе обычно определяются различные атрибуты xmlns. Они задают пространства имен XML, определяющие различные подчиненные элементы. Как минимум, элемент ‹definitions› должен указать пространство имен, где определены сами элементы WSDL (http://schemas.xmlsoap.org/wsdl). Для того чтобы быть полезным, открывающий дескриптор ‹definitions› должен, кроме того, указать пространства имен XML, определяющие простые типы данных WSDL, типы XML-схемы, элементы SOAP, а также целевое пространство имен. Например, вот как выглядит раздел ‹definitions› для нашего Web-сервиса калькулятора.

‹?xml version="1.0" encoding="utf-8"?›

**‹wsdl:definitions**xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"

xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"

xmlns-mime="http://schemas.xmlsoap.org/wsdl/mime/"

xmlns:tns="http://www.IntertechTraining.com/"

xmlns:s="http://www.w3.org/2001/XMLSchema"

xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"

xmlns:http="http://schemes.xmlsoap.оrg/wsdl/http/"

targetNamespace="http://www.IntertechTraining.com/"

xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"›

…

**‹/wsdl**:**definitions›**

В контексте корневого элемента вы можете найти пять подчиненных элементов. Общий вид WSDL-документа должен быть примерно таким.

‹?xml version="1.0" encoding="utf-8"?›

‹wsdl:definitions …›

 ‹wsdl:types›

  ‹!-- **Список типов, доступных для данного Web-сервиса -**-›

 ‹wsdl:/types›

 ‹wsdl:message›

  ‹!-- **Формат сообщений -**-›

 ‹wsdl:/message›

 ‹wsdl:portType›

  ‹!-- **Информация портов -**-›

 ‹wsdl:/portType›

 ‹wsdl:binding›

  ‹!-- **Информация связывания -**-›

 ‹wsdl:/binding›

 ‹wsdl:service›

  ‹!-– **Информация о самом Web-сервисе XML -**-›

 ‹wsdl:/service›

‹wsdl:/definitions›

Как и следует ожидать, каждый из этих подчиненных элементов будет содержать дополнительные элементы и атрибуты, уточняющие описание имеющихся возможностей. Давайте по очереди рассмотрим наиболее важные из допустимых узлов.

**Элемент ‹types›**

Сначала мы рассмотрим элемент ‹types›, который содержит описания всех типов данных, предлагаемых Web-сервисом. Вы, возможно, знаете, что язык XML сам определяет ряд "базовых" типов данных, и все они определены в рамках пространства имен XML http://www.w3.org/2001/XMLSchema (которое должно быть указано в контексте корневого элемента ‹definitions›). Возьмем, например, метод Subtract() нашего Web-сервиса калькулятора, имеющий два входных параметра целочисленного типа. В терминах WSDL тип System.Int32 среды CLR описывается в контексте элемента ‹complexType›.

‹s:еlement name= "Subtract"›

 <s:comlexType>

  ‹s:sequence›

   ‹s:element minOccurs="**1**" maxOccurs="**1**" name="**x**" type="**s:int**" /›

   ‹s:element minOccurs=''**1**" maxOccurs="**1**" name="**y**" type="**s:int**" */›*

*‹/*s:sequence›

 ‹/s:complexType›

‹/s:element›

Целое число, возвращаемое методом Subtract(), также описывается в рамках элемента ‹types›.

‹s:element name= "**SubtractResponse**"›

 ‹s:complexType›

  ‹s:sequence›

   ‹s:element minOccurs="**1**" maxOccurs*=*"**1**" name="**SubtractResult**" type="**s:int**"/›

  ‹/s:sequence›

*‹*/s:complexType›

‹/s:element›

Если вы имеете Web-метод, возвращающий или получающий пользовательские типы данных, они также появятся в контексте элемента ‹complexType›. Детали того, как с помощью Web-метода сделать доступными пользовательские типы данных .NET, мы рассмотрим позже. Для примера предположим, что вы определили Web-мeтод, возвращающий структуру с именем Point.

public struct Point {

 public int x;

 public int y;

 public string pointName;

}

WSDL-описание для этой "сложной структуры" будет выглядеть примерно так.

‹s:complexType name="**Point**"›

 ‹s:sequence›

  ‹s:element minOccurs="**1**" maxOccurs="**1**" name="**x**" type="**s:int**" */›*

  ‹s:element minOccurs="**1**'' maxOccurs="**1**" name="**y**" type= "**s:int**" /›

  ‹s:element minOccurs="**0**" maxOccurs="**1**" name="**рointName**" type="**s:string**" /›

 ‹/s:sequence›

‹/s:complexType›

**Элемент ‹message›**

Элемент ‹message› используется для определения формата обмена запросами и ответами данного Web-метода. Поскольку один Web-сервис позволяет передачу множества сообщений между отправителем и получателем, одному WSDL-документу позволяется определять множество элементов ‹message›. Как правило, в этих определениях используются типы, указанные в рамках элемента ‹types›.

Независимо от количества элементов ‹message›, определенных в документе WSDL, они обычно "присутствуют" парами. Первое определение представляет входной формат сообщения, а второе – выходной формат того же сообщения. Например, метод Subtract() Web-сервиса CalculatorWebService определяет следующие элементы ‹message›.

‹wsdl:message name="**SubtractSoapIn**"›

 ‹wsdl:part name="parameters" element="tns:Subtract" /›

‹/wsdl:message›

‹wsdl: message name="**SubtractSoapOut**"›

 ‹wsdl:part name="parameters" element="tns:SubtractResponse" /›

‹/wsdl:message›

Здесь вы видите только связь SOAP соответствующего сервиса. Как говорилось в начале этой главы, Web-сервисы XML могут вызываться с помощью SOAP или HTTP-методов GET и POST. Но если вы разрешите связь HTTP POST (соответствующие объяснения будут предложены позже), генерируемый WSDL-код должен продемонстрировать следующие данные ‹message›.

‹wsdl: message name="**SubtractHttpPostIn**"›

 ‹part name="n1" type="s:string" /›

 ‹part name="n2" type="s:string" /›

‹wsdl:/message›

‹wsdl:message name="**SubtractHttpPostOut**"›

 ‹part name="Body" element="s0:int" /›

‹wsdl:/message›

Элементы ‹message› сами по себе не слишком полезны. Однако на эти определения сообщений ссылаются другие части WSDL-документа.

**Замечание.**Не все Web-методы требуют и запроса, и ответа. Если Web-метод является "односторонним", для него необходим только элемент ‹message› запроса. Обозначить Web-метод, как односторонний, можно с помощью атрибута [SoapDocumentMethod].

**Элемент ‹portType›**

Элемент ‹portType› определяет различные связи, которые могут возникать между клиентом и сервером, и каждая такая связь представляется вложенным элементом ‹operation›. Несложно догадаться, что самыми типичными операциями здесь должны быть SOAP, HTTP GET и HTTP POST. Однако есть и другие операции. Например, односторонняя операция позволяет клиенту отправить сообщение данному Web-серверу, но не получить ответ (это похоже на вызов метода без ожидания возвращаемого значения). Операция "требование-ответ" позволяет серверу отправить, запрос во время ответа клиента (что можно рассматривать, как дополнение операции "запрос-ответ").

Чтобы проиллюстрировать формат необязательного вложенного элемента ‹operation›, рассмотрим WSDL-определение для метода Subtract().

‹wsdl portType name=**"CalculatorWebServiceSoap**"›

 ‹wsdl:operation name="**Subtract**"›

  ‹wsdl:input message="**tns:SubtractSoapIn**" /›

  ‹wsdl:output message="**tns:SubtractSoapOut**" */›*

*‹*/wsdl:operation›

‹wsdl:/portType›

Обратите внимание на то, как элементы ‹input› и ‹output› ссылаются на соответствующее имя сообщения, определенное в рамках элемента ‹message›. Если бы для метода Subtract() был разрешен HTTP-метод POST, вы бы увидели следующий дополнительный элемент ‹operation›.

‹wsdl:portType name="CalculatorWebServiceHttpPost"›

 <wsdl:operation name="**Subtract**" ›

  ‹wsdl:input message="**s0:SubtractHttpPostIn**" /›

  ‹wsdl:output message= "**s0:SubtractHttpPostOut**" /›

*‹*wsdl:/operation›

‹wsdl:/portType›

Наконец, учтите то, что если данный Web-метод описан с помощью свойства Description, элемент ‹operation› будет содержать вложенный элемент ‹documentation›.

**Элемент ‹binding›**

Этот элемент указывает точный формат обмена GET, POST и SOAP. Это самый "многословный" из всех элементов, содержащихся в контексте корневого элемента ‹definition›. Вот, например, определение элемента ‹binding› с описанием того, как вызывающая сторона может взаимодействовать с Web-методом MyMethod(). используя SOAP.

‹wsdl:binding name="СаlculatorWebServiceSoap12" type="**tns:CalculatorWebServiceSoap**"›

 ‹soap12:binding transport="**http://schemas.xmlsoap.org/soap/http**" */›*

  ‹wsdl:operation name= "**Subtract**"›

  ‹soap12:operation soapAction="**http://www.IntertechTraining.com/Subtract**" style="document" /›

  ‹wsdl:input›

   ‹soap12:body use="**literal**" /›

  ‹/wsdl:input›

  ‹wsdl:output›

   ‹soap12:body use="**literal**" /›

  ‹/wsdl:output›

 ‹/wsdl:operation›

‹/wsdl:binding›

**Элемент ‹service›**

Наконец, у нас есть элемент ‹service›, который указывает характеристики самого Web-сервиса (например, его URL). Главной задачей этого элемента является описание множества портов, открытых данным Web-сервером. Для этого элемент ‹services› может использовать любое число вложенных элементов ‹port› (не путайте их с элементом ‹portType›). Вот как выглядит элемент ‹service› для CalculatorWebService.

‹wsdl:service name=**"CalculatorWebService**"›

 ‹wsdl:documentation xmlns:wsdl**="http://schemas.xmlsoap.org/wsdl/"›**

**Чудесный Web-сервис калькулятора**

 ‹/wsdl:documentation›

 ‹wsdl:port name=**"CalculatorWebServiceSoap"**binding=**"tns**:**CalculatorWebServiceSoap"**›

  ‹soap:address location=**"http://localhost:1109/CalculatorWebService/ Service.asmx"**/›

 ‹/wsdl:port›

 ‹wsdl:port **name="CalculatorWebServiceSoap12"**binding= "**tns**:**CalculatorWebServiceSoap12**"›

  ‹soap12:address location=**"http://localhost:1109/CalculatorWebService/Service.asmx"**/›

 ‹/wsdl:port›

‹/wsdl:service›

Итак, как видите, WSDL-код, автоматически возвращаемый сервером ITS, не является сверхсложным, но, поскольку WSDL представляет собой грамматику на основе XML, этот код достаточно "многословен". Тем не менее, теперь вы должны лучше понимать роль WSDL, так что давайте рассмотрим немного подробнее протоколы связи Web-сервисов XML.

**Замечание.**Напомним, что пространство имен System.Web.Services.Description содержит множество типов, которые позволяют программно читать и обрабатывать "сырой" WSDL-код (можете проверить сами, если вас это интересует).

**Снова о протоколах связи Web-сервисов XML**

Строго говоря, Web-сервисы XML могут использовать для коммуникации любой RPC-протокол (например, DCOM или CORBA). Однако большинство Web-серверов встраивает соответствующие данные в тело HTTP-запроса и переправляет их адресату, используя для этого один из трех базовых способов связи (табл. 25.4).

Хотя каждый из подходов обеспечивает один и тот же результат (вызов Web-метода), от выбора протокола зависит то, какие типы параметров (и типы возвращаемых значений) могут пересылаться между заинтересованными сторонами. Протокол SOAP предлагает наибольшую гибкость, поскольку сообщения SOAP позволяют осуществлять обмен сложными типами данных (а также двоичными файлами) между вызывающей стороной и Web-сервисом XML. Однако для полноты давайте выясним роль стандартных HTTP-методов GET и POST.

**Таблица 25.4**. Режимы связи Web-сервисов XML

|  |  |
| --- | --- |
| **Режим связи** | **Описание** |
| HTTP-метод GET | В режиме обмена GET параметры добавляются к строке запроса данного URL |
| HTTP-метод POST | В режиме обмена POST данные встраиваются в заголовок HTTP-сообщения, а не добавляются к строке запроса |
| SOAP | SOAP является протоколом связи, определяющим правила передачи данных и вызова методов в сети с помощью XML |

**Связь HTTP GET и HTTP POST**

Хотя GET и POST кажутся привычными конструкциями, этот метод пересылки недостаточно гибок для обслуживания таких сложных элементов, как структуры и классы. При использовании SET и POST вы можете взаимодействовать с Web-методами, используя только типы, указанные в табл. 25.5.

**Таблица 25.5.**Типы данных, поддерживаемые методами GET и POST

|  |  |
| --- | --- |
| **Типы данных** | **Описание** |
| Перечни | GET и POST поддерживают передачу типов System.Enum.NET, поскольку эти типы представляются в виде статических строковых констант |
| Простые массивы | Вы можете использовать массивы любых примитивных типов |
| Строки | GET и POST осуществляют передачу любых числовых данных в виде строковых маркеров. *Строка*здесь на самом деле обозначает строковое представление среды CLR для таких примитивов, как Int16, Int32, Int64, Boolean*,*Single, Double, Decimal и т.д. |

По умолчанию HTTP-связь GET и POST не разрешена для удаленного вызова Web-сервисов XML. Однако HTTP-связь POST активизирована для вызова машиной локальных Web-сервисов (на самом деле именно этот режим использует автоматически генерируемая страница тестирования). Эти установки указываются в файле machine.config с помощью элемента ‹protocols›. Вот как выглядит соответствующий фрагмент

‹!-- **В файле machine. config -**-›

‹webServices*›*

 ‹protocols›

  ‹add name="HttpSoap1.2" /›

  ‹add name="HttpSoap" */›*

  ‹add name="Documentation" /›

**‹!-- HTTP GET/POST отключены! --›**

**‹!-- ‹add name="HttpPost''/› --›**

**‹!-- ‹add name="HttpGet"/› --›**

**‹!-- Используется страницей тестирования Web-сервиса**--›

  ‹add name="HttpPostLocalhost" /›

 ‹/protocols›

‹/webServiсes›

Чтобы снова разрешить использование HTTP-методов GET или POST для Web-сервиса, добавьте имена HttpPost и HttpGet в соответствующий локальный файл Web.config.

‹configuration›

 ‹system.web›

  ‹webServices›

   ‹protocols›

**‹add name="HttpPost"/›**

**‹add name="HttpGet"/›**

   ‹/protocols›

  ‹/webServices›

 ‹/system.web›

‹/configuration›

Снова напоминаем, что при использовании стандартных HTTP-методов GET и POST у вас нет возможности строить Web-методы, допускающие использование составных типов (например, DataSet ADO.NET или пользовательский тип структуры) в качестве параметров или возвращаемых значений. Для простых Web-сервисов это ограничение может быть вполне приемлемым. Однако при использовании связи SOAP вы можете строить гораздо более совершенные Web-сервисы XML.

**Связь SOAP**

Полный анализ возможностей SOAP выходит за рамки этого текста, однако следует понимать, что SOAP нельзя назвать специальным протоколом, который может использоваться наряду с другими существующими протоколами Интернет (HTTP, SMTP и др.). Общая задача SOAP, тем не менее, остается той же: обеспечить независимый от языка и платформы механизм вызова методов, использующих составные типы. Для этого SOAP преобразует каждый метод в сообщение SOAP.

Сообщение SOAP состоит из двух основных секций. Во-первых, есть конверт SOAP, который можно понимать, как абстрактный контейнер для соответствующей информации. Во-вторых, есть правила, которые используются для описания информации в сообщении (размещаемой в теле сообщения SOAP). Необязательная третья секция (заголовок SOAP) может использоваться для того, чтобы указать общую информацию, касающуюся самого сообщения (например, информацию безопасности или транзакции).

‹soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"›

 ‹soap:Header›

**‹!-- Необязательная информация заголовка --›**

 ‹/soap:Header›

 ‹soap:Body›

**‹!-- Информация вызова метода --›**

 ‹/soap:Body›

‹/soap:Envelope›

**Просмотр сообщения SOAP**

Хотя при создании Web-сервиcов XML в рамках платформы .NET от вас не требуется понимания всех деталей SOAP, вы можете увидеть формат сообщения SOAP дня каждого доступного Web-метода с помощью автоматически генерируемой страницы тестирования. Например, если щелкнуть на ссылке для метода Add() нашего CalculatorWebService, вы увидите следующий запрос SOAP 1.1.

‹soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/ "›

 ‹soap:Body›

  ‹Add xmlns="http://www.IntertechTraining.com"›

   ‹x›int‹/x›

   ‹y›int‹/y›

  ‹/Add›

 ‹/soap:Body›

‹/soap:Envelope›

Соответствующий ответ SOAP 1.1 выглядит так.

‹soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"›

 ‹soap:Body›

  ‹AddResponse xmlns="http://www.IntertechTraining.com "›

   ‹AddResult*›*int‹/AddResult›

  ‹/AddResponse›

 ‹/soap:Body›

‹/soap:Envelope›

**Утилита командной строки wsdl.exe**

Теперь, когда у вас есть базовые знания о WSDL и SOAP, давайте выясним, как с помощью такого инструмента командной строки, как wsdl.exe строить программы клиента, взаимодействующие с удаленными Web-сервисами XML. В сущности, wsdl.exe решает две важные задачи.

• Генерирование файла сервера, функционирующего в качестве каркаса для реализации Web-сервиса XML.

• Генерирование файла клиента, функционирующего в качестве агента удаленного Web-сервиса XML.

Утилита wsdl.exe поддерживает ряд флагов командной строки, список которых можно увидеть, указав при вызове этой утилиты опцию -? в командной строже. Описания некоторых аргументов wsdl.exe приводятся в табл. 25.6.

**Таблица 25.6.**Подборка опций wsdl.exe

|  |  |
| --- | --- |
| **Флаг командной строки** | **Описание** |
| /appsettingurlkey | Дает указание wsdl.exe создать агент, не использующий "жестко" заданные значения URL. Вместо этого класс агента будет настроен на чтение значений URL из файла \*.config клиента |
| /language | Указывает язык для использования в генерируемом классе агента: cs (C#; это значение используется по умолчанию), VB (Visual Basic .NET), JS (JScript), VJS (Visual J#) |
| /namespace | Указывает пространство имен для генерируемого агента или шаблона. По умолчанию сам тип в рамках определения пространства имен не определяется |
| /out | Указывает файл, в котором нужно сохранить программный код генерируемого агента. Если файл не указан, имя файла будет соответствовать имени Web-сервиса XML |
| /protocol | Указывает протокол, используемый в программном коде агента, по умолчанию это SOAP. Но можно также указать HttpGet или HttpPost, чтобы создать агент, использующий для взаимодействия HTTP-методы GET или POST |
| /serverInterface | Генерирует интерфейсные связи сервера для Web-сервиса XML на основе WSDL-документа |

**Замечание.**Флаг /server утилиты wsdl.exe в .NET 2.0 больше не используется. Теперь базовый программный код для сервера генерируется с помощью /serverlnterfасе.

**Преобразование WSDL-кода в серверный программный код Web-сервиса**

Одним из интересных вариантов использования утилиты wsdl.exe является генерирование серверного программного кода на основе WSDL-документа (с помощью опции /serverInterfасе). Очевидно, если вы начинаете разработку Web-сервиса XML с создания WSDL-документа, эта опция должна быть для вас очень важна. После того как файл исходного кода будет сгенерирован, вы получите хорошую исходную позицию для реализации каждого Web-метода.

Предположим, что вы создали WSDL-документ (CarBizObject.wsdl), в котором описывается единственный метод DeleteCar(), получающий на вход целое число и не возвращающий ничего. Этот метод предлагается Web-сервисом XML с именем CarBizObject, который может вызываться с использованием связи SOAP.

Чтобы сгенерировать серверный файл программного кода C# на основе этого WSDL-документа, откройте окно командной строки .NET и вызовите утилиту wsdl.exe с флатом /serverInterface, за которым должно следовать имя соответствующего WSDL-документа. Заметьте, что WDSL-документ может содержаться либо в локальном файле \*.wsdl:

wsdl /serverInterface CarBizObject.wsdl

либо получаться динамически по данному URL с помощью указания суффикса ?wsdl:

wsdl /serverInterface http://localhost/CarService/CarBizObject.asmx**?wsdl**

После того как wsdl.exe обработает соответствующие XML-элементы, вы получите описания интерфейсов для каждого Web-метода.

[System.Web.Services.WebServiceBindingAttribute(

 Name="CarBizObjectSoap",

 Namespace="http://IntertechTraining.com/")]

 public partial interface **ICarBizObjectSoap**{

 …

**void RemoveCar(int carID)**;

}

Используя эти интерфейсы, вы можете определить класс, реализующий различные методы Web-сервиса XML.

**Исходный код**. Файл CarBizObject.wsdl размещен в подкаталоге, соответствующем главе 25.

**Преобразование WSDL-кода в программный код агента для клиента**

Хотя это и нежелательно, но вполне возможно построить базовый программный код клиента, которым будет вручную открывать HTTP-соединение, строить SOAP-сообщения, вызывать Web-методы и выполнять обратную трансляцию поступающего XML-потока в типы данных CTS. Намного более предпочтительным подходом оказывается использование wsdl.exe для генерирования класса агента, который будет представлять Web-методы, определенные данным файлом \*.asmx.

Для этого укажите (как минимум) имя генерируемого файла агента (с помощью флага /out) и место размещения WSDL-документа. По умолчанию wsdl.exe генерирует программный код агента на языке C#. Однако если вы хотите иметь программный код агента на другом языке .NET, вы можете использовать флаг /language. Следует также знать, что по умолчанию wsdl.exe генерирует программный код агента, предполагающего связь с удаленным Web-сервисом XML с помощью SOAP. Чтобы созданный агент использовал HTTP-метод GET или POST. следует указать соответствующий протокол связи с помощью /protocol.

Другим важным моментом в отношении генерирования программного кода агента с помощью wsdl.exe является то, что этому инструменту действительно требуется *WSDL-документ*Web-сервиса XML, а не просто файл с именем \*.asmx. С учетом этого следует понимать, что если для разработки и тестирований Web-сервиса вы используете WebDev.WebServer.exe, то перед генерированием программного кода агента для клиента вы, скорее всего, захотите скопировать содержимое проекта в виртуальный каталог IIS.

Для примера предположим. что вы создали новый виртуальный каталог IIS (CalcService), содержащий данные проекта CalculatorService. После этого вы можете сгенерировать программный код агента клиента так.

wsdl /out:proxy.cs http://localhost/CalcService/Secrvice.asmx?wsdl

В качестве замечания подчеркнем, что wsdl.exe не определяет пространство имен .NET для упаковки генерируемых типов C#. если вы не укажете в командной строке флаг /n.

wsdl /out:proxy.cs /n:CalculatorClient http://localhost/CalcService/ Service.asmx?wsdl

**Программный код агента**

Если открыть сгенерированный файл агента, вы найдете там тип, который получается из System.Web.Services.Protocols.SoapHttpClientProtocol (если, конечно, вы не указали другой протокол связи с помощью опции /protocol).

public partial class **CalculatorWebService**:

 System.Web.Services.Protocols.SoapHttpClientProtocol {

 …

}

Этот базовый класс определяет ряд членов, используемых в рамках реализации типа агента. Описания некоторых из этих членов предлагаются в табл. 25.7.

**Таблица 25.7.**Основные члены типа SoapHttpClientProtocol

|  |  |
| --- | --- |
| **Унаследованные члены** | **Описание** |
| BeginInvoke() | Метод, инициирующий асинхронный вызов Web-метода |
| CancelAsync() | Метод (новый в .NET 2.0), отменяющий асинхронный вызов  метода Web-сервиса XML, если вызов еще не завершен |
| EndInvoke() | Метод, завершающий асинхронный вызов Web-метода |
| Invoke() | Метод для синхронного вызова метода Web-сервиса |
| InvokeAsync() | Метод (новый в .NET 2.0), предлагающий более  предпочтительный вариант асинхронного вызова метода Web-сервиса |
| Proxy | Свойство, получающее или устанавливающее информацию агента для  запроса Web-сервиса через брандмауэр |
| Timeout | Свойство, получающее или устанавливающее значение времени ожидания  (в миллисекундах) для синхронных вызовов |
| Url | Свойство, получающее или устанавливающее базовое значение URL сервера  для запросов |
| UserAgent | Свойство, получающее или устанавливающее значение для заголовка  пользовательского агента в запросах |

**Конструктор, заданный по умолчанию**

Заданный по умолчанию конструктор агента "жестко" определяет значение URL удаленного Web-сервиса и запоминает это значение в наследуемом свойстве Url.

public CalculatorWebService() {

 this.Url = "http://localhost/CalcServicе/Service.asmx";

}

Очевидным недостатком такого подхода является то, что при переименовании или перемещении Web-сервиса XML класс агента приходится обновлять и перекомпилировать. Для построения более гибкого типа агента wsdl.exe предлагает использовать флаг /appsettingurlkey (который можно сократить до /urlkey). Если указать в командной строке этот флаг, конструктор агента будет содержать программную логику для чтения URL с помощью ключа, содержащегося в файле \*.config клиента.

wsdl /out:proxy.cs /n:СаlcClient /urlkey:**CalcUrl**http://localhost/CalcService/Serviсе.asmx?wsdl

Если теперь проверить конструктор агента, заданный по умолчанию, вы обнаружите следующий программный код (заметьте, что если подходящий ключ не будет найден, в качестве резервного будет использоваться заданное конкретное значение URL).

public CalculatorWebService() {

 string urlSetting = System.Configuration.ConfigurationManager.AppSettings["CalcUrl"];

 if ((urlSetting != null)) {

  this.Url = urlSetting;

 } else {

  this.Url = "http://localhost./CalcService/Service.asmx";

 }

}

Соответствующий файл app.config на стороне клиента будет примерно таким.

‹?xml version="1.0" encoding="utf-8"?›

‹configuration›

 ‹appSettings›

  ‹add key**="CalcUrl"**value="http://localhost/CalcService/Service.asmx" /›

 ‹/appSettings›

‹/configuration›

**Поддержка синхронного вызова**

Генерируемый агент определяет также поддержку синхронного вызова Web-методов. Например, синхронный вариант метода Subtract() реализуется так.

public int Subtract(int x*,*int y) {

 object[] results = this.invoke("Subtract", new object[] {x, y});

 return ((int)(results[0]));

}

Обратите внимание на то, что вызывающая сторона передает два параметра, "упакованные" в массив System.Object. Используя динамическое связывание, метод Invoke() передаст эти аргументы методу вычитания, размещенному по указанному адресу URL. По завершении этого (блокирующего) вызова будет обработан поступающий XML-код и результат будет возвращен вызывающей стороне в виде System.Int32 после соответствующего преобразования.

**Поддержка асинхронного вызова**

Поддержка асинхронного вызова Web-методов в .NET 2.0 сильно изменилась по сравнению с .NET 1.x. По своему предыдущему опыту вы можете знать, что агенты .NET 1.1 использовали методы BeginXXX()/EndXXX() для вызова Web-методов во вторичном потоке выполнения. Рассмотрите, например, следующие методы BeginSubtract() и EndSubtract().

public System.IAsyncResult BeginSubtract(int x, int y, System.AsyncCallback callback, object asyncState) {

 return this.BeginInvoke("Subtract", new object[] {x, y}, callback, asyncState);

}

public int EndSubtract (System.IAsyncResult asyncResult) {

 object[] results = this.EndInvoke(asyncResult);

 return ((int) (results[0]));

}

Хотя wsdl.exe все еще генерирует эти знакомые методы Begin/End, в .NET 2.0 они считаются устаревшими, поскольку заменены новыми методами XXXAsync().

public void SubtractAsync(int x, int y) {

 this.SubtractAsync(x, y, null);

}

Новые методы XXXAsync() (как и связанный с ними метод CancelAsync()) работают в паре с автоматически генерируемым вспомогательным методом (являющимся перегруженной версией некоторого специального метода XXXAsync()), который обрабатывает асинхронные операции, используя синтаксис событий C#. Если рассмотреть программный код агента, вы увидите, что wsdl.exe генерирует (для каждого Web-метода) пользовательский делегат, пользовательское событие и пользовательский класс "event args", чтобы получить соответствующий результат.

**Создание приложения клиента**

Теперь, когда вы лучше понимаете внутреннюю композицию генерируемого агента, давайте попытаемся его использовать. Создайте новое консольное приложение с именем CalculatorClient, добавьте в проект файл proxy.cs с помощью выбора Project→Add Existing Item из меню и добавьте ссылку на компоновочный блок System.Web.Services.dll. Затем измените метод Main() так, как предлагается ниже.

class Program {

 static void Main(string[] args) {

  Console.WriteLine("\*\*\*\*\* Забавы c агентами WS \*\*\*\*\*\n");

**// Создание агента.**

  CalculatorWebService ws = new CalculatorWebService();

**// Синхронный вызов метода Add().**

  Console.WriteLine("10 + 10= {0}", ws.Add(10, 10));

**// Асинхронный вызов метода Subtract с помощью**

**// нового подхода .NET 2.0 на основе событий.**

  ws.SubtractCompleted += new SubtractCompleteEventHandler(ws\_SubtractCompleted);

  ws.SubtractAsync(50, 45);

**// Продолжение работы консоли для гарантии получения**

**// результата вычитания**.

  Console.RеаdLine();

 }

 static void ws\_SubtractCompleted(object sender, SubtractCompletedEventArgs e) {

  Console.WriteLine("Baш ответ: {0} ", e.Result);

 }

}

Обратите внимание на то, что новая логика асинхронного вызова в .NET 2.0 непосредственно отображается в синтаксис событий C#, который, согласитесь, является более аккуратным по сравнению с использованием методов BeginXXX()/EndXXX(), интерфейса IAsyncResult и делегата AsyncCallback.

**Исходный код**. Проект CalculatorClient размещен в подкаталоге, соответствующем главе 25.

**Генерирование программного кода агента в Visual Studio 2005**

Утилита wsdl.exe, конечно, предлагает целый ряд аргументов командной строки, которые позволяют контролировать результат генерирования класса агента, но Visual Studio 2005 позволяет быстро сгенерировать файл агента, используя диалоговое окно Add Web Reference (Добавление Web-ссылки), которое можно вызвать из Меню Project. Как видно из рис. 26.6, в этом окне вы можете получить ссылки на существующие Web-сервисы XML, размещенные в самых разных местах.

**Замечание**. Диалоговое окно Add Web Reference не позволяет ссылаться на Web-сервисы XML, которые обслуживаются WebDev.WebServer.exe.

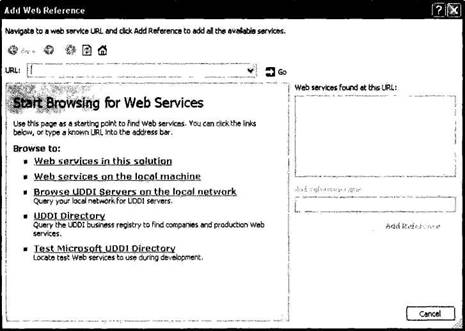


Рис. 25.6. Диалоговое окно Add Web Reference

Обратите внимание на то, что вы имеете возможность не только получить список Web-сервисов на своей локальной машине, но и запросить различные каталоги UDDI (соответствующие вопросы будут обсуждаться в конце главы). Так или иначе, в результате ввода в строку URL подходящего значения, указывающего на действительный файл \*.wsdl или \*.asmx, вы добавите в проект новый класс агента. Заметьте, что пространство имен агента (зависящее от URL источника) будет вложено в рамки пространства имен вашего клиента .NET. Так, если для клиента с именем MyClientApp добавляется ссылка на Web-сервис, размещенный на вашей локальной машине, вы должны указать C#-директиву using следующего вида.

using MyClientApp.localhost;

**Замечание.**В Visual Studio 2005 диалоговое окно Add Web Reference автоматически либо добавляет в проект новый файл app.config, содержащий значения URL ссылок на Web-сервисы XML, либо обновляет уже существующий.

**Доступ к пользовательским типам Web-методов**

В заключительном примере этой главы мы с вами выясним, как строить Web-сервисы, предлагающие доступ к пользовательским типам, а также к более "экзотическим" типам из библиотек базовых классов .NET. Для примера мы создадим новый Web-сервис XML, который будет способен обрабатывать массивы, пользовательские типы и объекты DataSet ADO.NET. Сначала создайте новый Web-сервис XML с именем CarSalesInfoWS, размещенный в виртуальном каталоге IIS.

**Доступ к массивам**

Создайте Web-метод GetSalesTagLines(), который возвращает массив строк, представляющих данные текущих продаж различных автомобилей, и метод SortCarMakes(), который позволит вызывающей стороне передать массив несортированных строк, чтобы обратно получить новый массив отсортированных строк.

[WebService(Namespace="http://IntertechTraining.com/", Description="Автомобильный Web-сервис", Name="CarSalesInfoWS")]

[WebServiceBinding(ConformsTo=WsiProfiles.BasicProfile1\_1)]

public class Service: System.Web.Services.WebService {

 [WebMethod(Description="Получение рекламных скидок")]

 public string[] GetSalesTagLines() {

  string[] currentDeals = {

   "Цены на Colt снижены на 50%",

   "Bce BMW комплектуются 8-канальным звуком",

   "Caravan бесплатно… спросите у дилера!"

  };

  return currentDeals;

 }

 [WebMethod(Description = "Сортировки списка марок")]

 public string[] SortCarMakes(string[] theCarsToSort) {

  Array.Sort(theCarsToSort);

  return theCarsToSort;

 }

}

**Замечание**. Страница тестирования, генерируемая с помощью DefaultWsdlHelpGenerator.aspx, не может вызывать методы, использующие в качестве параметров массивы типов.

**Доступ к структурам**

Протокол SOAP позволяет передачу XML.предcтавлений пользовательских типов данных (таких как классы и структуры). Web-сервисы XML используют тип XmlSerializer для преобразования типа в XML-код (см. главу 17, где имеется более подробная информация по этому поводу). Напомним, что XmlSerializer:

• не выполняет сериализацию приватных данных: для сериализации используются только открытые поля и свойства;

• требует, чтобы каждый позволяющий сериализацию класс имел конструктор, заданный по умолчанию;

• не требует использования атрибута [Serializable].

С учетом сказанного, наш следующий Web-метод будет возвращать массив структур SalesInfoDetails, определенных следующим образом.

**// Пользовательский тип.**

public struct SalesInfoDetails {

 public string info;

 public DateTime dateExpired;

 public string Url;

}

Другим интересным моментом в отношении XmlSerializer является то, что этот тип позволяет осуществлять многослойный контроль представления типа. По умолчанию сериализация структуры SalesInfoDetails выполняется путем преобразования каждого поля данных поля в уникальный XML-элемент.

‹SalesInfoDetails›

 ‹info›Цены на Colt снижены на 50'%!‹/info›

 ‹dateExpired›2004-12-02T00:00:00.0000000-06:00‹/dateExpired›

 ‹Url›http://www.CarsRUs.com‹/Url›

‹/SalesInfoDetails›

Чтобы изменить поведение, предлагаемое по умолчанию, вы можете в определения своих типов добавить атрибуты, определенные в пространстве имен System.Xml.Serialization (снова см. главу 17).

public struct SalesInfoDetails {

 public string info;

**[XmlAttribute]**

 public DateTime dateExpired;

 public string Url;

}

В результате будет получено следующее XML-представление данных.

‹SalesInfoDetails dateExpired="2004-12-02T00:00:00"›

 ‹info›Цены на Colt снижены на 50%!‹/info›

 ‹Url›http://www.CarsRUs.com‹/Url›

‹/SalesInfoDetails›

Реализация GetSalesInfoDetails() возвращает заполненный массив этой пользовательской структуры.

[WebMethod(Description="Get details of current sales")]

public SalesInfoDetails[] GetSalesInfoDetails() {

 SalesInfoDetails[] theInfo = new SalesInfoDetails[3];

 theInfo[0].info = "Цены на Colt снижены на 50%!";

 theInfo[0].dateExpired = DateTime.Parse("12/02/04");

 theInfo[0].Url = "http://www.CarsRUs.com";

 theInfo[1].info = "Все BMW комплектуются 8-канальным звуком";

 theInfo[1].dateExpired = DateTime.Parse("8/11/03");

 theInfo[1].Url = "http://www.Bmws4U.com";

 theInfo[2].info = "Caravan бесплатно… спросите у дилера!";

 theInfo[2].dateExpired = DateTime.Parse("12/01/09");

 theInfo[2].Url = "http://www.AllPinkVans.com";

 return theInfo;

}

**Доступ к типам DataSet ADO.NET**

Чтобы завершить создание нашего Web-сервиса XML, вот вам еще един Web-метод, который возвращает DataSet с данными таблицы Inventory базы данных Cars, созданной при изучении ADO.NET в главе 22.

**// Получение списка всех машин из таблицы Inventory.**

[WebMethod(Description = "Возвращает список машин из таблицы Inventory базы данных Cars")]

pubic **DataSet**SetCurrentInventory() {

**// Заполнение DataSet данными таблицы Inventory.**

 SqlConnection sqlConn = new SqlConnection();

 sqlConn.ConnectionString = "data source=localhost;initial catalog=Cars;uid=sa;pwd=";

 SqlDataAdapter myDA= new SqlDataAdapter("Select \* from Inventory", sqlConn);

 DataSet ds = new DataSet();

 myDA.Fill(ds*,* "Inventorу");

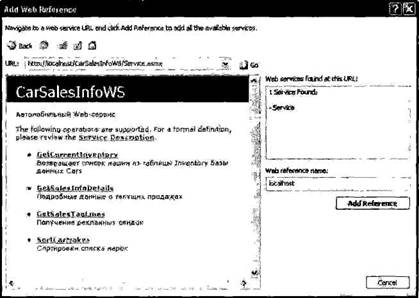
 return ds;

}

**Исходный код**. Файлы примера CarsSalesInfoWS размещены в подкаталоге, соответствующим главе 25.

**Клиент Windows Forms**

Чтобы проверить работу нового Web-сервисa XML, создайте приложение Windows Forms и укажите в нем ссылку на CarsSalesInfoWS, используя диалоговое окно Add Web Reference в Visual Studio 2005 (рис. 25.7).



Pиc. 25.7. Добавление ссылки на CarsSalesInfoWS

Теперь просто используйте генерируемый агент для вызова доступных Web-методов. Вот один из возможных вариантов реализации формы.

using CarsSalesInfoCLient.localhost;

…

namespace CarsSalesInfoClient {

public partial class MainWindow: Farm {

 private CarSalesInfoWS ws = new CarSalesInfoWS();

 …

 private void MainWindow\_Load(object sender, EventArgs e) {

**// Привязка к таблице.**

  inventoryDataGridView.DataSource = ws.GetCurrentInventory().Tables[0];

 }

 private void btnGetTagLines\_Click(object sender, EventArgs e) {

  string[] tagLines = ws.GetSalesTagLines();

  foreach (string tag in tagLines) listBoxTags.Items.Add(tag);

 }

 private void btnGetAllDetails\_Click (object sender, EventArgs e) {

  SalesInfoDetails[] theSkinny = ws.GetSalesInfoDetails();

  foreach (SalesInfoDetails s in theSkinny) {

   string d = string.Format("Info: {0}\nURL:(1}\nExpiration Date:{2}", s.info, s.Url, s.dateExpired);

   MessageBox.Show(d, "Details");

  }

 }

}

На рис. 25.8 показан результат тестового запуска приложения.

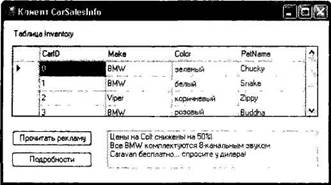


Рис. 25.8. Клиент CarsSalesInfo

**Представление типов на стороне клиента**

Когда клиент устанавливает ссыпку на Web-сервис, предоставляющий доступ к пользовательским типам, файл класса агента получает определения для каждого отрытого пользовательского типа. Так, если вы посмотрите на представление клиента для SalesInfoDetails (в генерируемом файле Reference.cs), вы увидите, что каждое поле инкапсулировано в строго типизованное свойство (также обратите внимание на то, что этот тип теперь определен, как класс, а не как структура).

[System.SerializableAtttribute()]

[System.Xml.Serialization.XmlTypeAttribute(Namespace="http://IntertechTraining.com/")]

public partial class **SalesInfoDetails**{

 private string infoField;

 private string urlField;

 private System.DateTime dateExpiredField;

 public string info {

  get { return this.infoField; }

  set { this.infoField = value; }

 }

 public string Url {

  get { return this.urlField; }

  set { this.urlField = value; }

 }

 [System.Xml.Serialization.XmlAttributeAttribute()]

 public System.DateTime dateExpired {

  get { return this.dateExpiredField; }

  set { this.dateExpiredField = value; }

 }

}

При этом, конечно, следует понимать, что подобно случаю удаленного взаимодействия .NET, типы, сериализация которых выполняется по сети в формате XML, не сохраняют логику реализации. Поэтому, если структура SalesInfoDetails поддерживала ряд открытых методов, генератор агента учесть это не сможет (прежде всего, потому, что эти методы не отражены в документе WSDL!). Однако если вы установите компоновочный блок клиента, который будет содержать программный код реализации типа клиента, вы сможете использовать программную логику типа. Но при этом требуется, чтобы соответствующая машина обеспечивала поддержку .NET.

**Исходный код.**Проект CarSaleslnfoClient размещен в подкаталоге, соответствующем главе 25.

**Стандарт поиска и взаимодействия (протокол UDDI)**

Тот факт, что первый типичный шаг клиента при обращении к удаленному Web-серверу оказывается темой заключительного раздела этой главы, может казаться немного странным. Причиной этой несообразности является то, что процесс проверки существования (или отсутствия) данного Web-сервиса с помощью UDDI является не только необязательным, но во многих случаях вообще ненужным.

Пока Web-сервисы XML не стали де-факто стандартом распределенных вычислений, большинство Web-сервисов используется компаниями в жесткой связи с конкретным поставщиком. При этом и компания, и поставщик уже хорошо знают один другого, поэтому у них не возникает никакой необходимости запрашивать UDDI-сервер для выяснения вопроса о существовании Web-сервиса. Однако если создатель Web-сервиса XML пожелает открыть свободный доступ к функциональным возможностям своего Web-сервиса, этот Web-сервис можно внести в UDDI-каталог.

UDDI (Universal Description, Discovery and Integration – универсальное описание, поиск и взаимодействие, стандарт UDDI) является инициативой, позволяющей разработчикам Web-сервисов "выставить" коммерческий Web-сервис в общеизвестном хранилище. Независимо от того, что вы могли сейчас подумать, UDDI не является технологией, поддерживаемой исключительно фирмой Microsoft. На самом деле IBM и Sun Microsystems имеют не меньшую заинтересованность в успexe инициативы UDDI. Как и следует ожидать, UDDI-каталоги обслуживаются многими поставщиками. Например, посвященный UDDI официальный Web-сайт Microsoft доступен по адресу http://uddi.microsoft.com. Официальный Web-сайт UDDI (http://www.uddi.org) предлагает множество "белых книг" и SDK, которые позволяют строить внутренние UDDI-серверы.

**Взаимодействие с UDDI в Visual Studio 2005**

Вспомните о том, что диалоговое окно Add Web Reference позволяет не только получить список всех Web-сервисов XML, размещенных на вашей локальной машине (а также по известному URL), но и предъявить запрос к UDDI-серверу. Как правило, вы имеете на выбор следующие варианты.

• Выполнить обзор UDDI-сервера во внутренней сети вашей компании.

• Выполнить обзор спонсируемого фирмой Microsoft производственного UDDI-сервера.

• Выполнить обзор спонсируемого фирмой Microsoft тестового UDDI-сервера.

Предположим, что вы строите приложение, которое должно получать текущий прогноз погоды на основе почтового индекса. Вашим первым шагом должен быть запрос к UDDI-каталогу с вопросом следующего содержания.

• Есть ли у вас Web-сервисы, имеющие отношение к погодным данным?

В том случае, когда UDDI-сервер имеет список Web-сервисов, связанных с прогнозом погоды, вы получите список всех зарегистрированных URL, предлагающих возможности, соответствующие вашему запросу. С помощью этого списка вы сможете выбрать подходящий вам Web-сервис и, в конечном итоге, получить WSDL-документ с описанием функциональных возможностей, cвязанных с обработкой погодных данных.

Для примера создайте новый проект консольного приложения и активизируйте диалоговое окно Add Web Reference. Затем выберите ссылку Test Microsoft UDDI Directory, которая приведет вас к тестовому UDDI-серверу Microsoft. После этого укажите слово **weather**(погода) в качестве критерия поиска. После выполнения запроса UDDI-каталогом вы получите список всех соответствующих Web-сервисов XML. Обнаружив Web-сервис XML который подойдет вам для использования в разработке, добавьте ссылку на этот сервис в свой проект. Как вы и должны ожидать, "сырой" WSDL-код будет проанализирован соответствующим инструментом разработки, и вы получите соответствующий агент на языке C#.

**Замечание.**Следует понимать, что центр тестирования является не более чем центром тестирования. Не удивляйтесь, если вы обнаружите там множество недействительных ссылок. При запросе производственных UDDI-серверов их URL оказываются гораздо более надежными, поскольку компаниям обычно приходится платить за то, чтобы они присутствовали в соответствующем списке.

**Резюме**

В этой главе были рассмотрены базовые компоненты Web-сервисов и основные этапы их построения в пределах платформы .NET. Глава начинается с рассмотрений пространств имен (и базовых типов в этих пространствах имен), используемых при создании Web-сервисов. Вы узнали, что для разработки Web-сервиса в .NET требуется лишь немногим большее, чем применение атрибута [WebMethod] к каждому члену, который вы хотите сделать доступным в рамках типа, представляющего данный Web-cервис XML. Вы также можете (но это необязательно) сделать соответствующие типы производными от System.Web.Services.WebService, чтобы (среди прочего) получить доступ к свойствам Application и Session. В связи с основной темой обсуждения этой главы были также рассмотрены три взаимосвязанные ключевые технологии – это механизм поиска (UDDI), язык описания (WSDL) и протоколы связи (GET, POST или SOAP).

После создания любого числа членов с атрибутами [WebMethod] вы можете начать взаимодействие с Web-сервисом посредством промежуточного агента. Такой агент может быть сгенерирован с помощью утилиты wsdl.exe, а сам агент может использоваться клиентом подобно любому другому типу C#. Альтернативой инструменту командной строки wsdl.exe является использование Visual Studio 2005, где аналогичные возможности предлагает диалоговое окно Add Web Reference.