Основной язык технологии Microsoft .net

Си Шарп

Создание приложений для Windows

В. В. Лабор

Си Шарп

Создание приложений для Windows

Минск
Харвест
2003

УДК 681.3.06
ББК 32.97
Л 39

Лабор В. В.

Л 39 Си Шарп: Создание приложений для Windows/ В. В. Лабор.— Мн.: Харвест, 2003. - 384 с.

ISBN 985-13-1405-6.

Не так давно компания Microsoft известила весь мир о создании новой технологии .NET. Эта технология выводит программирование на новый уровень развития. Специально для нее компания Microsoft разработала язык C# (Си Шарп), который является новейшим на сегодняшний день языком программирования. Он сочетает в себе преимущества уже существующих языков программирования и дополняет их удобными механизмами работы с технологией .NET .

Эта книга* позволит вам в короткие сроки ознакомиться с основными возможностями C#. Вы сможете изучить синтаксис языка, типы данных, классы, свойства, методы, атрибуты и многое другое. Также в книге подробно рассмотрена методика создания приложений для Windows. Все описанные в книге возможности языка подчеркиваются многочисленными примерами. Прочитав эту книгу, вы сможете с легкостью приступить к созданию собственных приложений на языке C#.

Книга будет интересна как новичкам, так и имеющим опыт программистам.

УДК 681.3.06
ББК 32.97

ISBN 985-13-1405-6

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В последнее время C и C++ становятся наиболее используемыми языками при разработке коммерческих и бизнес-приложений. Эти языки устраивают многих разработчиков, но в действительности не обеспечивают должной продуктивности разработки. К примеру, процесс написания приложения на C++ часто занимает гораздо больше времени, чем разработка эквивалентного приложения на Visual Basic. Сейчас существуют языки, увеличивающие продуктивность разработки за счет потери в гибкости, которая так привычна и необходима программистам на C/C++. Подобные решения весьма неудобны для разработчиков и нередко предлагают значительно меньшие возможности. Эти языки также не ориентированы на взаимодействие с появляющимися сегодня системами и очень часто не соответствуют существующей практике программирования для Web. Многие разработчики хотели бы использовать современный язык, который позволял бы писать, читать и сопровождать программы с простотой Visual Basic и в то же время давал мощь и гибкость C++, обеспечивал доступ ко всем функциональным возможностям системы, взаимодействовал с существующими программами и легко работал с возникающими Web-стандартами.

Учитывая все подобные пожелания, Microsoft разработала новый язык — C#. Он имеет массу преимуществ: простота, объектная ориентированность, типовая защищенность, «сборка мусора», поддержка совместимости версий и многое другое. Данные возможности позволяют быстро и легко разрабатывать приложения. При создании C# его авторы учитывали достижения многих других языков программирования: C++, C, Java, Visual Basic и т.д. Надо заметить, что поскольку C# разрабатывался что называется «с нуля», у его авторов была возможность не переносить в него все неудачные особенности любого из предшествующих языков. Особенно это касается проблемы совместимости с предыдущими версиями. В результате получился действительно простой, удобный и современный язык, который по мощности не уступает C++, но существенно повышает продуктивность разработок.

Ввиду высокой объектной ориентированности, язык C# великолепно подходит для быстрого конструирования различных компонентов — от высокоуровневой бизнес-логики до системных приложений, использующих низкоуровневый код. Также следует отметить, что C# является и Web-ориентированным — с помощью простых встроенных конструкций языка ваши компоненты легко превратятся в Web-сервисы, к которым можно будет обращаться из Интернета, используя любой язык на любой операционной системе. Дополнительные возможности и преимущества C# перед другими языками приносит использование современных Web-технологий, таких как: XML (Extensible Markup Language) и SOAP (Simple Object Access Protocol). Удобные методы для разработки Web-приложений позволяют программистам, владеющим навыками объектно-ориентированного программирования, легко освоиться в разработке Web-сервисов.

Кому предназначена эта книга

Эта книга для тех, кто решил начать разработку на C# и .NET. Для того чтобы приступить к изучению ее, вам не обязательно знать какой-либо другой язык программирования. Книга содержит описание необходимых понятий в области объектно-ориентированного программирования, описание синтаксиса языка, типов данных, классов, методов. Это позволит вам быстро овладеть ключевыми понятиями программирования на языке C#. Хотя, несомненно, знания в области C, C++ или Java вам бы пригодились.

Основная направленность книги — это обучение созданию приложений для Windows на языке C#. Однако знания, которые вы получите, помогут вам с легкостью овладеть технологией создания приложений для Web.

Прочитав эту книгу, вы сможете быстро и легко приступить к разработке собственных приложений на языке C#.

Как вы понимаете, одно издание не может стать исчерпывающим источником информации по такому языку программирования, как C#, Для повышения своего уровня владения языком вы можете обратиться к другим книгам или ресурсам сети Интернет.

Требования к системе

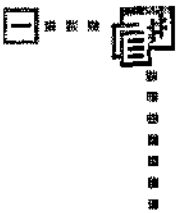
Для эффективной работы с данной книгой вы должны иметь возможность компилировать и исполнять программы на C#. Для этого ваш компьютер должен удовлетворять следующим требованиям:

- Microsoft .NET Framework SDK;
- Microsoft Visual C# или Microsoft Visual Studio .NET;
- Microsoft Windows NT 4.0, Windows 2000 или Windows XP.

Чтобы написанные на C# программы можно было запускать на других компьютерах, на них должна быть установлена исполняющая среда .NET (dotnetfx.exe). Этот пакет поставляется вместе с .NET Framework SDK и Visual Studio .NET.

Для успешной установки Visual Studio.NET вам понадобятся: процессор — Pentium II, 450 МГц, лучше Pentium III, 600 МГц, свободное место на жестком диске — 500 Мбайт на системном диске, 3 Гбайт на установочном диске, операционная система: Windows XP, Windows 2000 и Windows NT 4.0, оперативная память (в зависимости от операционной системы):

- Windows NT 4.0 Workstation — 64 Мбайт, лучше 96 Мбайт;
- Windows NT 4.0 Server — 160 Мбайт, лучше 192 Мбайт;
- Windows 2000 Professional — 96 Мбайт, лучше 128 Мбайт;
- Windows 2000 Server — 192 Мбайт, лучше 256 Мбайт;
- Windows XP Professional — 160 Мбайт, лучше 192 Мбайт.



Раздел I. Основные положения

1. ЯЗЫК C# И КАРКАС .NET

Появление нового языка программирования, в данном случае C#, всегда вызывает интерес у программистов, даже если они не собираются в ближайшее время использовать его в своей работе. В этой главе речь пойдет об основных идеях, заложенных в новый язык.

КАКИЕ ЦЕЛИ СТОЯЛИ ПЕРЕД РАЗРАБОТЧИКАМИ C#

Андерс Хиджисберг, который возглавил в Microsoft работу по созданию языка C#, следующим образом определил стоявшие перед ними цели:

- создать первый компонентно-ориентированный язык программирования семейства C/C++;
- создать объектно-ориентированный язык, в котором любая сущность представляется объектом;
- упростить C++, сохранив его мощь и основные конструкции.

Главное новшество связано с заявленной компонентной ориентированностью языка. Компоненты позволяют решать проблему модульного построения приложений на новом уровне. Построение компонентов обычно определяется не только языком, но и платформой, на которой этот язык реализован.

Платформа .NET — многоязыковая среда, открытая для свободного включения новых языков, создаваемых не только Microsoft, но и другими фирмами. Все языки, включаемые в платформу .NET, должны опираться на единый каркас, роль которого играет .NET Framework. Это серьезное ограничение, одновременно являющееся и важнейшим достоинством.

.NET Framework И БИБЛИОТЕКА КЛАССОВ

В основе большинства приложений, создаваемых в среде VC++ 6.0, лежал каркас приложений (Application Framework), ключевую роль в котором играла библиотека классов MFC. Когда создавался новый проект MFC — EXE, ActiveX или DLL, из каркаса приложений выбирались классы, необходимые для построения проекта с заданными свойствами. Выбранные классы определяли каркас конкретного приложения.

Каркас .NET также содержит библиотеку классов (Class Library). Она служит тем же целям, что и любая библиотека классов, входящая в каркас. Библиотека включает множество интерфейсов и классов, объединенных в группы по тематике. Каждая группа задается пространством имен

(namespace), корневое пространство имен называется System. Классы библиотеки связаны отношением наследования. Все классы являются наследниками класса System.Object. Для классов библиотеки, равно как и для классов в языке C#, не определено множественное наследование.

СРЕДА ВЫПОЛНЕНИЯ Common Language Runtime

Библиотека классов — это статическая составляющая каркаса. В .NET Framework есть и динамическая составляющая — система, определяющая среду выполнения,— CLR (Common Language Runtime). Роль этой среды весьма велика — в ее функции входит управление памятью, потоками, безопасностью, компиляция из промежуточного байт-кода в машинный код и многое другое. Важный элемент CLR — это мощный механизм «сборки мусора» (garbage collector), управляющий работой с памятью.

Язык C# в полной мере позволяет использовать все возможности CLR. Код, создаваемый на C#, в принципе безопасен. Иными словами, если вы будете придерживаться установленной методики программирования на C#, то вряд ли сможете написать код, способный привести к неправильному обращению с памятью.

В целом, следует отметить следующие моменты:

- среда .NET Framework, задающая единый каркас многоязыковой среды разработки приложений, во многом ориентирована на компонентное программирование. Она оказывает несомненное влияние на все языки, поддерживающие эту технологию;
- разработчики .NET Framework просто не могли не создать новый язык программирования, который в полной мере отвечал бы всем возможностям .NET Framework, не неся на себе бремени прошлого. Таким языком и стал язык C#;
- главным достоинством языка C# можно назвать его согласованность с возможностями .NET Framework и вытекающую отсюда компонентную ориентированность.

СТРУКТУРА ПРОГРАММЫ

Программа на C# состоит из одного или нескольких файлов. Каждый файл может содержать одно или несколько пространств имен. Каждое пространство имен может содержать вложенные пространства имен и типы, такие как классы, структуры, интерфейсы, перечисления и делегаты — функциональные типы. При создании нового проекта C# в среде Visual Studio выбирается один из 10 возможных типов проектов, в том числе Windows Application, Class Library, Web Control Library, ASP.NET Application и ASP.NET Web Service. На основании сделанного выбора автоматически создается каркас проекта.

C# и C++

Авторы всячески подчеркивают связь языков C# и C++. Но есть серьезные различия, касающиеся синтаксиса, семантики отдельных конструкций.

Указатели и управление памятью

В языке C++ работа с указателями занимает одно из центральных мест. Нормальный стиль программирования на C# предполагает написание безопасного кода, а это значит — никаких указателей, никакой адресной арифметики, никакого управления распределением памяти. Возможность работы с указателями в духе C++ ограничена «небезопасными» блоками. Небезопасный код для C#-программистов будет скорее исключением, чем правилом. Это позволит меньше отвлекаться на отслеживание корректности работы программы с памятью, уделяя больше внимания функциональной части программы.

Наследование и шаблоны

В языке C# не реализованы такие важные для C++ моменты, как множественное наследование и шаблоны. Множественное наследование в C# возможно только для интерфейсов.

Типы данных

В языке C# появилась принципиально новая классификация типов, подразделяющая типы на значимые и ссылочные. Как следствие, применяются разные способы работы с объектами этих типов. В языке устранена разница между переменными и объектами. Все переменные в C# — тоже объекты, которые имеют единого предка — класс System.Object.

Структуры

В языке C++ структуры подобны классу, за небольшими исключениями. В C# разница между структурой и классом более существенна: структуры не могут иметь наследников, классы относятся к ссылочным типам, а структуры — к значимым.

Массивы

В языке C# имеется возможность как объявлять классические массивы, так и работать с массивами при помощи встроенных классов. Работа с массивами в C# более безопасна, поскольку выход за границы массива контролируется (при условии использования безопасного кода).

Классы

Следует отметить различия в подходах к сокрытию свойств класса. В C++ такое понятие, как свойство, вообще отсутствовало. В C# введены процедуры-свойства get и set, аналогичные тому, как это сделано в языке VB. Синтаксис обращения к свойствам класса в C# аналогичен синтаксису обращения к данным.

Синтаксические и семантические детали

В C# оператор switch не требует задания break для прерывания операции. Булевы переменные в языке C# имеют два значения, вместо них нельзя использовать целочисленные переменные, как это принято в C++. В C# точка используется всюду, где в C++ применяются три разных символа — «.», «::», «->».

C# И Java

Вначале о сходстве, которое, несомненно, носит принципиальный характер. Можно отметить общую ориентацию на Web и на выполнение приложений в распределенной среде, близкую по духу среду выполнения с созданием промежуточного байт-кода, синтаксическую и семантическую близость обоих языков к языку C++.

Создание Web-ориентированной платформы .NET — это веление времени. Microsoft просто обязана была создать ее, независимо от существования платформы Java. Хотя, несомненно, разработчики .NET учитывали, что им придется конкурировать с уже существующей платформой.

.NET Framework как единый каркас многоязыковой среды создания приложений — принципиальное новшество, не имеющее аналога. По своей сути такое решение автоматически избавляет программистов от многих проблем создания совместно работающих компонентов, написанных на разных языках.

Появление новой технологии .NET Framework автоматически определило целесообразность создания и нового языка программирования, в полной мере согласованного с идеями, заложенными в ней.

Следует отметить два важных отличия в деталях реализации C# и Java. В среде выполнения Java промежуточный байт-код интерпретируется, в то время как в CLR — компилируется, что повышает эффективность исполнения. По некоторым данным, время исполнения уменьшается в десятки раз. И Java, и C# многое взяли от общего предка — языка C++. Однако авторы новых языков разделились в заимствовании многих конструкций. Например, разработчики Java отказались от перечислений, тогда как в C# этот тип был не только сохранен, но и развит. В C# сохранена также возможность перегрузки операторов. Кроме того, в C# добавлены многие полезные нововведения.

2. ОБЗОР СРЕДЫ РАЗРАБОТКИ Visual Studio .NET

Visual Studio .NET КАК НОВАЯ КОНЦЕПЦИЯ Microsoft

Можно сказать, что жизненный путь платформы .NET по-настоящему начался с появлением на рынке комплекса средств разработки нового поколения Visual Studio .NET. Его официальное представление и продажа первых коробочных версий состоялись в феврале 2002 г. При этом желающие могли получить бета-версии замечательного продукта еще раньше. Важность Visual Studio .NET для всей концепции Microsoft .NET объяснима: успешное продвижение платформы напрямую зависит от наличия широкого круга прикладных программ, позволяющих работать с .NET-технологией.

Сразу после появления первой публичной бета-версии Visual Studio .NET в конце 2000 г. стала очевидной серьезность намерений Microsoft относительно своей новой платформы. Разработчики сразу поняли, что их ожидает не просто очередная модификация инструментария, а серьезное изменение подхода к созданию приложений. В публикациях не утихали споры о преимуществах и недостатках новой платформы, появилось много противников очередного нововведения Microsoft. Однако большинство ждало выхода в свет официальной версии продукта.

Дебаты о новшествах Visual Studio .NET утихли с появлением в начале лета 2001 г. второй бета-версии пакета. Стало понятно, что Microsoft будет настойчиво продвигать предложенные новинки. Программистам не оставалось ничего другого, как начать осваивать новую технологию, перейти на платформу другого поставщика, упорно продолжать работать в среде Visual Studio 6.0, дожидаясь последних дней его жизни, либо менять профессию.

Следует заметить, что речь идет не о простом обновлении версии используемого продукта, а о радикальном изменении технологии разработки. Хотя в Visual Studio .NET и сохранился рабочий номер версии 7.0, правильнее говорить о Visual Studio .NET 1.0.

ВОЗМОЖНОСТИ СРЕДЫ РАЗРАБОТКИ Visual Studio .NET

Даже если вы уже работали с какой-либо средой разработки от компании Microsoft, то будете приятно удивлены появлением новых возможностей в среде Visual Studio .NET. Если нет, то вам предстоит многое освоить. Новая среда содержит огромный набор «полезностей», призванных

упростить жизнь программисту. Я не буду делать обзор всех возможностей среды Visual Studio .NET, однако основные элементы следует отметить. Кроме того, многие пункты меню и управляющие окна будут описаны далее по тексту книги.

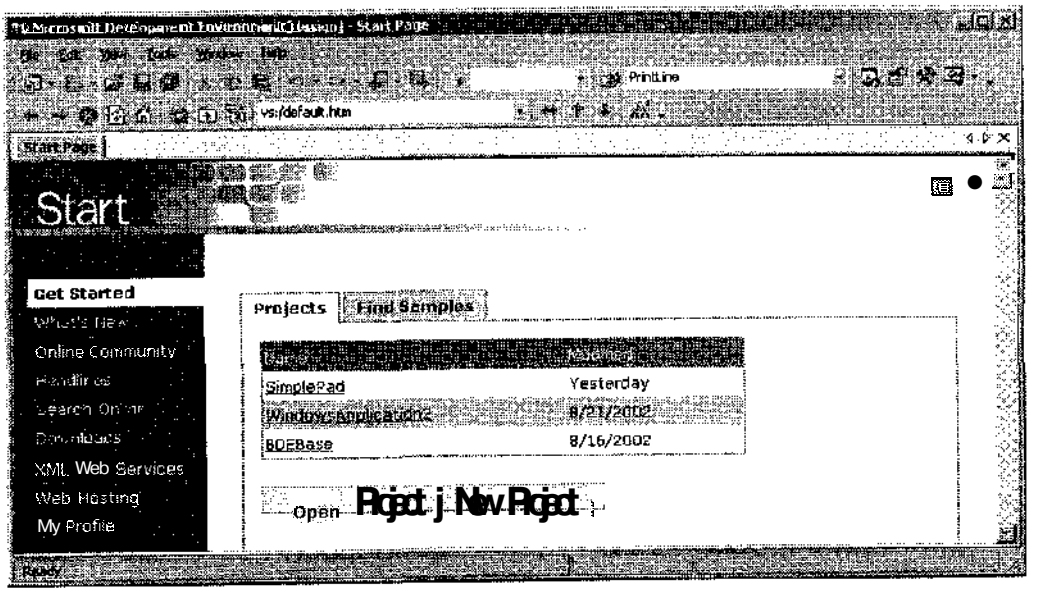
Стартовая страница

Давайте запустим Visual Studio .NET. Для этого после установки программы выберите пункт меню *Пуск/Программы/Microsoft Visual Studio .NET/Microsoft Visual Studio .NET*. На экране появится среда с ее стартовой страницей Visual Studio Home Page, изображенной на рис. 2.1. В правом окне виден список последних выполненных проектов, а также кнопки, с помощью которых можно открыть ранее созданный проект или создать новый. Закладка Find Samples поможет по ключевым словам выполнять поиск подходящих примеров, записанных в соответствующих разделах установочного каталога.

В левой части находится список дополнительных команд. С помощью My Profile можно настроить представление среды в соответствии с вашими личными пожеланиями. Ссылка What's New познакомит вас с новшествами VS .NET. Остальные команды представляют собой ссылки на различные Интернет-ресурсы. К сожалению, их нельзя пополнять самостоятельно.

Если вы случайно закрыли стартовую страницу или хотите открыть ее в процессе работы над проектом, просто выберите пункт меню *Help/Show Start Page*.

Рис. 2.1. Стартовая страница.



Создание проекта

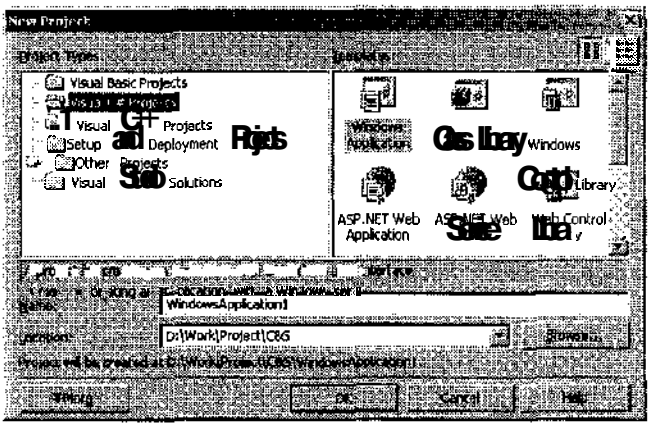


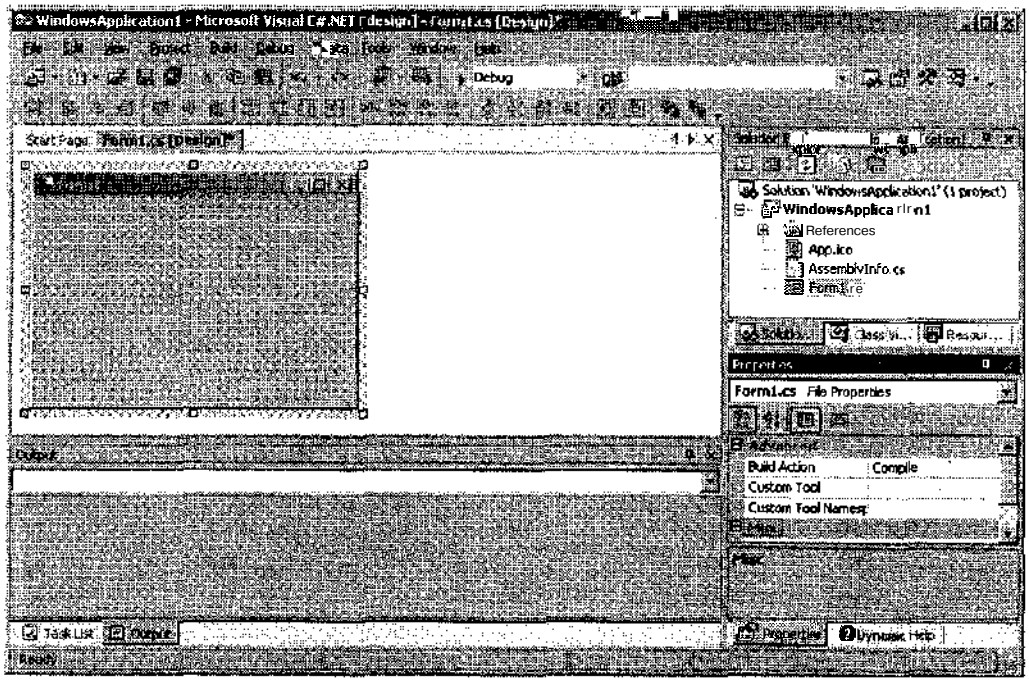
Рис. 2.2. Окно создания нового проекта.

Настало время сказать, что Visual Studio.NET — это не только среда для разработки приложений на языке C#. Visual Studio .NET позволяет создавать приложения на языках VB, C#, C++, формировать Setup (установочный пакет) ваших приложений и многое другое.

Для того чтобы реально увидеть, как создается новый проект в Visual Studio .NET, выберите пункт меню *File/New/Project...* После его вызова появится окно, аналогичное изображенному на рис. 2.2.

Здесь можно выбрать нужный вам язык программирования (в левой части окна) или какой-то специальный мастер создания приложений —

Рис. 2.3. Главное окно среды Visual Studio .NET.



этот список может пополняться инструментами независимых разработчиков. Поскольку наша книга посвящена программированию на C#, выберите пункт *Visual C# Project.*

В правой части окна нужно указать тип создаваемого вами проекта. Это может быть Windows-приложение (Windows Application), приложение для Интернет (ASP.NET), консольное приложение (Console Application) и некоторые другие. Выберите в левой части окна пункт *Windows Application.* Кроме того, вы можете указать название создаваемого проекта и путь к каталогу, в котором он будет располагаться. Нажмите *OK.*

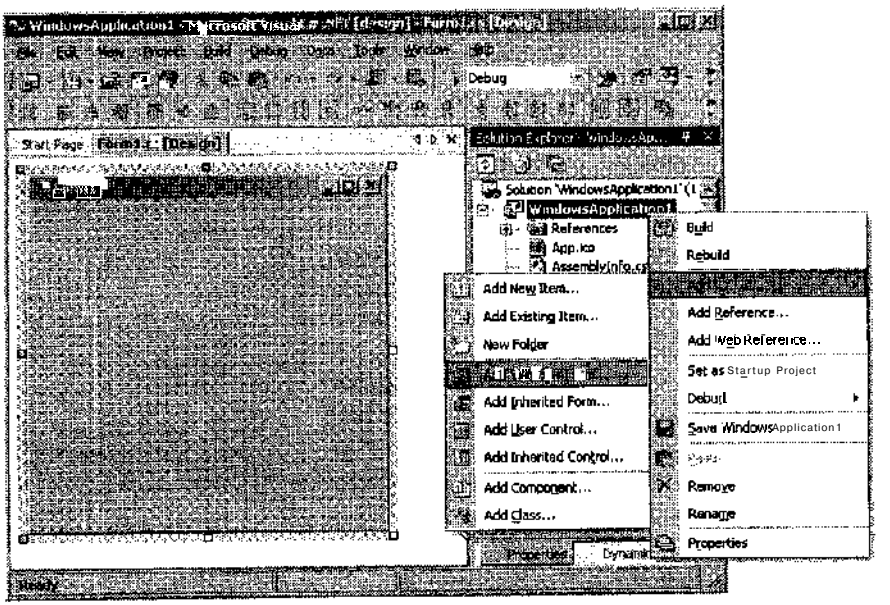
Теперь вы увидите основные части визуальной среды разработки проекта. Они изображены на рис. 2.3. В центре находится главное окно для создания визуальных форм и написания кода. Справа размещается окно Solution Explorer для управления вашими проектами, Class View для обзора всех классов и окно свойств Properties Explorer.

Solution Explorer

Solution Explorer позволяет управлять компонентами, включенными в ваш проект. Например, для того чтобы добавить в него новую форму, просто выберите в контекстном меню, открывающемся по щелчку правой кнопки мыши, пункт *Add/Add Windows Form* (см. рис. 2.4).

Речь о других пунктах этого меню пойдет в следующих главах. Кроме контекстного меню проекта существует еще ряд контекстных меню, позволяющих управлять отдельными элементами проекта. Так, чтобы переключиться из окна дизайнера в окно кода проекта, выберите в контек-

Рис. 2.4. Контекстное меню управления проектом.



стном меню для Forml пункт *View Code*. Соответственно, чтобы переключиться назад — *View Designer*. Контекстное меню есть у каждого элемента дерева проекта. Использование контекстного меню — это быстрый интерактивный способ навигации по проекту. Пожалуйста, запомните это!

Class View

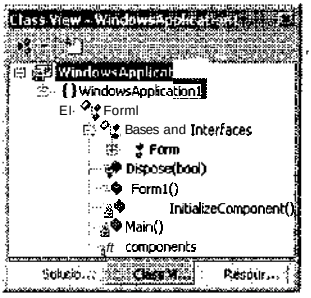


Рис. 2.5. Class View

Это окно позволяет перемещаться по всем элементам программного проекта, включая отдельные процедуры (см. рис. 2.5). При этом с помощью Class View можно добавлять новые методы, классы, данные. Все эти возможности будут доступны вам благодаря контекстным меню Class View. При этом заметьте, что каждый элемент дерева проекта имеет свой тип контекстного меню. Поэкспериментируйте с Class View. Это позволит вам более уверенно чувствовать себя в среде Visual Studio .NET. Если окно Class View отсутствует на экране, выберите пункт меню View/Class View.

Properties Explorer

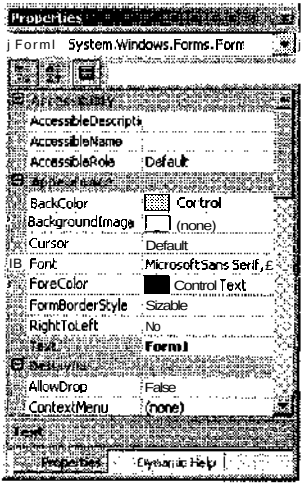


Рис. 2.6. Properties Explorer. Закладка «Свойства»

Это окно позволит вам работать со свойствами форм и их компонентов. Properties Explorer содержит список всех свойств выбранного в текущий момент компонента. Например, если вы выберите форму созданного нами приложения, окно Properties Explorer примет вид, представленный на рис. 2.6.

Здесь представлены два списка: список всех свойств формы (с правой стороны) и их значений (с левой стороны). При этом значения свойств могут быть представлены в любой форме, например, как текстовое поле, как выпадающий список допустимых значений, как окно выбора цвета и т. д. Если вы измените значение свойства по умолчанию, то оно будет выделено жирным цветом. В этом случае контроль за вносимыми вами в проект изменениями станет более наглядным.

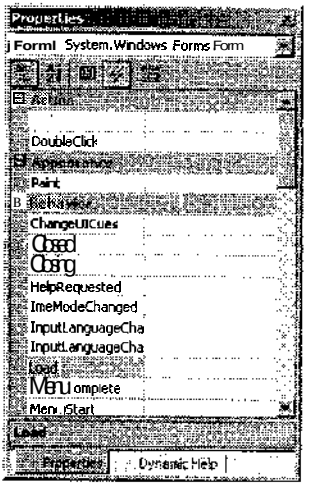


Рис. 2.7. Properties Explorer. Закладка «События»

Кроме того, Properties Explorer позволяет сортировать свойства либо по алфавиту, либо по принадлежности к определенной группе.

Второй важной задачей, которую выполняет Properties Explorer, является управление событиями. Для того чтобы переключиться на закладку событий, нажмите кнопочку с изображением молнии вверху окна (см. рис. 2.7).

Окно событий позволит вам настраивать реакцию вашей формы или компонента на различные действия со стороны пользователя или операционной системы, например создать обработчик событий от мыши или клавиатуры. В левой части окна содержится список всех доступных событий, а в правой — имен методов, обрабатывающих события. По умолчанию список методов пуст. Вы можете добавить новый обработчик, вписав имя метода в соответствующую ячейку, либо создать обработчик с именем по умолчанию, щелкнув два раза по ячейке левой кнопкой мыши.

Toolbox

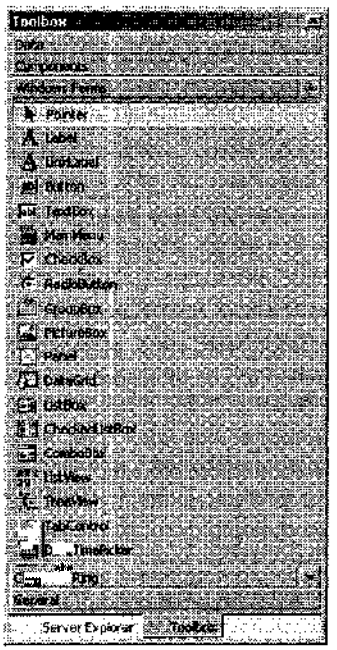


Рис. 2.8. Инструментальная панель (Toolbox)

Это окно содержит Windows Forms компоненты, которые вы можете разместить на своей форме. Если такого окна в вашем Visual Studio нет, выберите в главном меню пункт *View/ Toolbox* (см. рис. 2.8).

Окно визуально отображает наиболее часто используемые .NET компоненты для создания приложений Windows. Toolbox имеет несколько закладок: Data, Components, Windows Forms, Clipboard Ring и General.

Все, кроме Clipboard Ring и General, содержат компоненты, которые можно перетянуть мышью на форму. Закладка *Windows Forms* включает визуальные элементы управления, такие как кнопки, списки, деревья. Закладка *Data* посвящена базам данных. Закладка *Components* содержит невизуальные компоненты, наиболее представительным среди которых является Timer.

Clipboard Ring отображает содержимое буфера обмена за последние N операций копирования или вырезания. Для вставки содержимого, которое было скопировано в буфер обмена несколько операций назад, просто щелкните два раза левой кнопкой мыши по требуемой строке.

Визуальные свойства вспомогательных окон

Нельзя не отметить замечательные свойства всех вышеописанных окон. Во-первых, все они могут «прилипать» к любой стороне главного окна Visual Studio .NET (правда этим сейчас уже никого не удивишь). А во-вторых, пря-

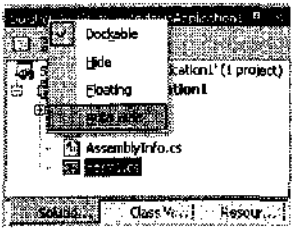


Рис. 2.9. Режим автоматического исчезновения окна с экрана

таться при потере активности. Для того чтобы наделить этим замечательным свойством, например, Solution Explorer, выберите в контекстном меню этого окна пункт Auto Hide (см. рис. 2.9.) или нажмите соответствующую кнопку рядом с кнопкой заголовка «Закрыть».

Сейчас, если Solution Explorer потеряет активность (например, вы сделаете активной форму), тогда его окно спрячется в прилежащую панель (см. рис. 2.10).

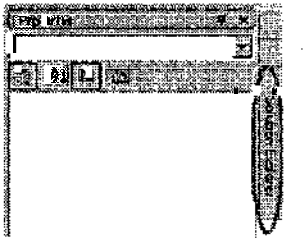


Рис 2.10. панель отображения скрыпых окон

Чтобы вернуть окно в первоначальное состояние, просто щелкните левой кнопкой мыши по соответствующему названию В панели.

Меню и панель инструментов

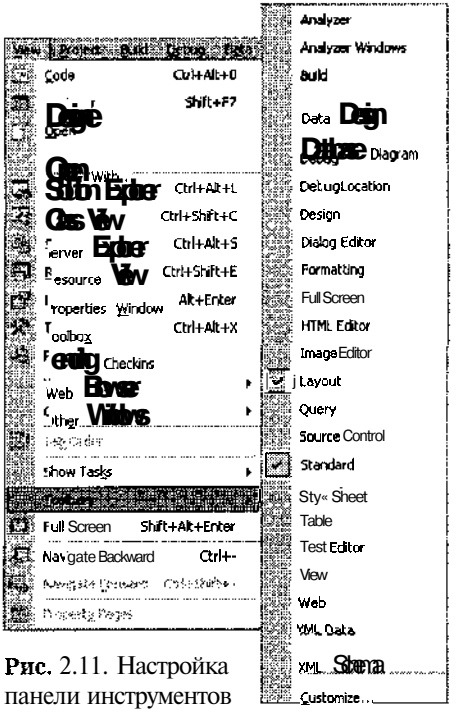


Рис. 2.11. Настройка панели инструментов

Все действия, которые вы можете выполнять в среде Visual Studio .NET, располагаются в главном меню. Главное меню имеет контекстную зависимость от текущего состояния среды, то есть содержит различные пункты в зависимости от того, чем вы сейчас занимаетесь и в каком окне находитесь.

Кроме того, большинство пунктов меню продублированы в панели инструментов. Visual Studio .NET имеет множество панелей инструментов. Вы можете включить или выключить панель инструментов при помощи меню *View/ Toolbars* (см. рис. 2.11).

Те панели инструментов, которые уже открыты, помечены в меню «птичками». Вы также можете создавать собственные панели инструментов, воспользовавшись пунктом этого же меню *Customize*.

Главное меню Visual Studio .NET

Меню Visual Studio .NET находится в верхней части среды. В меню есть все команды, предназначенные для выполнения действий над элементами проектов Visual Studio .NET. Пункты меню бьшают командными и группо-

выми (содержащими другие пункты меню). Название каждого группового пункта меню отражает содержащиеся в нем команды. Например, меню «File» содержит команды, предназначенные для работы с файлами проекта. Некоторые пункты меню включают вложенные пункты с более подробными командами. Например, команда «New» из меню «File» показывает меню выбора типов файлов. Наиболее часто употребляемые пункты меню имеют «горячие» клавиши. Так, для создания нового файла нужно нажать клавиши CTRL+N. Рассмотрим основные пункты главного меню Visual Studio .NET.

Меню File

Название команды	«Горячие» клавиши	Назначение команды
New >		
Project	Ctrl+Shift+N	Создать новый проект.
File	Ctrl+N	Создать новый файл.
Blank Solution	—	Создать новое решение.
Open >		
Project	—	Открыть созданный ранее проект.
Project For Web...	Ctrl+Shift+O	Открыть созданный ранее проект по сети.
File	Ctrl+O	Открыть отдельные файлы (как принадлежащие проекту, так и не связанные с ним).
File For Web...	—	Открыть отдельные файлы (как принадлежащие проекту, так и не связанные с ним по сети).
Close	—	Закрыть текущий открытый проект.
Add New Item...	Ctrl+Shift+A	Добавить в проект новый элемент.
Add Existing Item...	Alt+Shift+A	Добавить в проект уже существующий элемент.
Open Solution...	—	Открыть существующий проект решения.
Close Solution	—	Закрыть существующий проект решения.
Save Selected Items...	Ctrl+S	Сохранить содержимое активной страницы (название этого пункта меню изменяется в зависимости от названия открытой страницы).
Save Selected Items As...	—	Сохранить содержимое активной страницы под другим именем (название этого пункта меню изменяется в зависимости от названия открытой страницы).

Название команды	«Горячие» клавиши	Назначение команды
Save All	**Ctrl+Shift+S**	Сохранить все измененные в проекте файлы.
Page Setup ...	—	Настройка параметров страницы при печати.
Print...	**Ctrl+P**	Печать.
Resent Files...	—	Список последних редактируемых файлов (выбранное из списка откроется).
Resent Project...	—	Список последних редактируемых проектов (выбранное из списка приведет к закрытию текущего проекта и открытию выбранного).
Exit	—	Закрыть проект и выйти из среды.

Меню Edit

Название команды	«Горячие» клавиши	Назначение команды
Undo	**Ctrl+Z**	Отменить последние шаги редактирования.
Redo	**Ctrl+Y**	Повторить последние отмененные шаги редактирования.
Cut	**Ctrl+X**	Вырезать выделенный фрагмент в буфер.
Copy	**Ctrl+C**	Скопировать выделенный фрагмент в буфер.
Paste	**Ctrl+V**	Вставить из буфера фрагмент в текущее положение курсора или выделенную форму.
Delete	Del	Удалить выделенный фрагмент.
Select All	Ctrl+Y	Выделить все строки в текущем файле или все компоненты на форме.
Find and Replace...	—	Работа с поиском и автоматической заменой фрагментов текста программы.
Find	**Ctrl+F**	Найти заданный фрагмент в файле.
Replace	**Ctrl+H**	Найти заданный фрагмент в файле и заменить его на другой.
Find in Files ...	**Ctrl+Shift+F**	Найти заданный фрагмент во всех файлах текущего проекта с возможностью задания типов файлов для поиска.

Название команды	«Горячие» клавиши	Назначение команды
Replace in Files...	Ctrl+Shift+H	Найти заданный фрагмент во всех файлах текущего проекта и заменить его на другой (существует возможность задания типа файлов для поиска).
Find Symbol	Ctrl+Shift+Y	Найти во всех доступных файлах ключевой символ, переменную, имя функции и т.д.
Go To ...	Ctrl+G	Перейти на строку с заданным номером в текущем файле.
Insert File As Text ...	—	Вставить выбранный файл в текущее место курсора (файл вставляется в текстовом виде).
Advanced	—	Дополнительные возможности по редактированию текста программы.
Tabify Selection	—	Заменить в выделенном фрагменте текста идущие подряд пробелы на табуляции, если число пробелов больше, чем длина табуляции.
Untabify Selection	—	Заменить в выделенном фрагменте текста табуляции на идущие подряд пробелы.
Make Uppercase	Ctrl+Shift+U	Привести все символы в выделенном фрагменте текста к верхнему регистру.
Make Lowercase	Ctrl+U	Привести все символы в выделенном фрагменте текста к нижнему регистру.
Delete Horizontal White Space	Ctrl+K; Ctrl+/	Заменить несколько идущих подряд пробелов на один.
View White Space	Ctrl+W	Сделать видимыми пробелы и табуляции.
Word Wrap	Ctrl+R	Убрать горизонтальную полосу прокрутки.
Comment Selection	Ctrl+K; Ctrl+C	Закомментировать выделенные строки.
Uncomment Selection	Ctrl+K; Ctrl+U	Раскомментировать выделенные строки.
Bookmarks	—	Работать с закладками для перемещения по тексту программы и поиска нужных фрагментов.
Toggle Bookmark	Ctrl+K; Ctrl+K	Создать закладку на текущей строке.

Название команды	«Горячие» клавиши	Назначение команды
Next Bookmark	Ctrl+K; Ctrl+N	Перейти на следующую по тексту закладку.
Previous Bookmark	Ctrl+K; Ctrl+P	Перейти на предыдущую по тексту закладку.
Clear Bookmarks	Ctrl+K; Ctrl+L	Удалить все закладки.
Outliling	—	Работать со свойством «раскрытия» содержимого фрагмента (+) в окне редактора текста программы.
Toggle Outlining Expansion	Ctrl+M; Ctrl+M	Раскрыть/закрыть содержимое текущего фрагмента, где стоит курсор.
Toggle All Outlining	Ctrl+M; Ctrl+L	Закрыть/раскрыть до уровня выше раскрытые фрагменты файла.
Stop Outlining	Ctrl+M; Ctrl+P	Убрать с экрана возможность работы с фрагментами текста программы.
Collapse Do Definitions	Ctrl+M; Ctrl+O	Открыть на экране возможность работы с фрагментами текста программы.
IntelliSense	—	Дополнительные возможности по работе с классами и переменными.
List Members	Ctrl+J	Вызвать динамический список членов класса.
Parameter Info	Ctrl+Shift+Space	Отобразить список параметров.

Меню View

Название команды	«Горячие» клавиши	Назначение команды
Code	Ctrl+Alt+O	Перейти в файл редактирования текста программы.
Designer	Shift+F7	Перейти в редактор форм.
Solution Explorer	Ctrl+Alt+L	Открыть панель просмотра свойств Solution.
Class View	Ctrl+Shift+C	Открыть панель просмотра классов и переменных проекта.
Server Explorer	Ctrl+Alt+S	Открыть панель просмотра данных SQLServer.
Resource View	Ctrl+Shift+E	Открыть панель просмотра ресурсов проекта.

Название команды	«Горячие» клавиши	Назначение команды
Toolbox	Ctrl+Alt+X	Открыть панель с набором компонент по работе с формой.
Web Browser	—	Открыть окно Web-просмотра (эта панель включает стартовую страницу, настройку профайла и др.).
Other Windows	—	Работа с отображением дополнительных панелей для удобства работы.
Macro Explorer	Alt+"	Открыть панель обзора макросов (в правой части экрана).
Object Browser	Ctrl+Alt+J	Открыть панель обзора объектов текущего проекта.
Task List	Ctrl+Alt+K	Открыть панель списка задач.
Command Window	Ctrl+Alt+A	Открыть окно команд.
Output	Ctrl+Alt+O	Открыть окно результатов.
Show Tasks	—	Настройки для работы с Task List.
Toolbars	—	Работа с настройкой вида и функционального наполнения Toolbars среды разработки.
Full Screen	Shift+Alt+Enter	Развернуть окно редактора на весь экран. Для возврата повторите операцию еще раз.
Navigate Backward	Ctrl+ -	Переместиться в предыдущее открытое окно.
Navigate Forward	Ctrl+ Shift+ -	Вернуться назад в окно, из которого перешли при помощи операции "Navigate Backward".

Меню Project

Название команды	«Горячие» клавиши	Назначение команды
Add Windows Form	—	Добавить в проект новую форму "Windows Form".
Add Inherited Form	—	Добавить в проект новую форму "Inherited Form".
Add User Control	—	Добавить в проект новую форму "User Control".
Add Inherited Control	—	Добавить в проект новую форму "Inherited Form".

Название команды	«Горячие» клавиши	Назначение команды
Add Component	—	Добавить в проект новый "Component Class".
Add Class	—	Добавить в проект новый класс в отдельном файле.
Add New Item	Ctrl+Shift+A	Добавить в проект новый элемент (по умолчанию создает новый класс).
Add Existing Item	Shift+Alt+A	Добавить в проект новый элемент из ранее созданного проекта (соответствующие файлы копируются в каталог текущего проекта).
Exclude From Project	—	Исключить выбранный элемент из проекта.
Include In Project	—	Включить выбранный элемент в проект.
Show All Files	—	Отобразить в Solution Explorer все файлы, включенные в проект.
Add Reference	—	Добавить ссылку.
Add Web Reference	—	Добавить Web-ссылку.
Set as Start Up Project	—	Установить текущий выбранный проект как активный для запуска.

Меню Build

Название команды	«Горячие» клавиши	Назначение команды
Build Solution	F7	Откомпилировать файл.
Rebuild Solution	—	Перекомпилировать файл после изменений.
Build <имя проекта>	—	Откомпилировать весь текущий проект.
Rebuild <имя проекта>	—	Перекомпилировать весь текущий проект после изменений.
Bach Build ...	—	Построить необходимые проекты.
Configuration Manager	—	Настроить конфигурации построения проектов.

Меню Debug

Название команды	«Горячие» клавиши	Назначение команды
Windows\Breakpoints	**Ctrl+Alt+B**	Отобразить окно параметров установленных точек останова в проекте.
Start	F5	Запустить проект на выполнение.
Restart	Ctrl+F5	Остановить выполнение программы, перекомпилировать и запустить на выполнение заново.
Break All	Ctrl+Shift+Break	Временно приостановить выполнение программы.
Continue	F5	Продолжить выполнение после временной остановки программы.
Stop Debugging	Shift+F5	Остановить выполнение программы.
Detach All	Shift+F5	Открепить все прикрепленные к проекту процессы.
Start Without Debugging	Ctrl+ F5	Запустить проект на выполнение без возможности отладки.
Processes	Ctrl+ F5	Список всех процессов.
Step Into	**F11**	Пошаговое выполнение программы.
Step Over	F10	Выполнение программы до текущей позиции курсора.
New Breakpoint	**Ctrl+B**	Установить точку останова на строке текущей позиции курсора.
Clear All Breakpoint	Ctrl+Shift+F9	Удалить все установленные точки останова в проекте.
Disable All Breakpoint		Сделать все установленные точки останова в проекте неактивными.

Меню Tools

Название команды	«Горячие» клавиши	Назначение команды
Debug Processes	**Ctrl+Alt+P**	Отладить один из запущенных процессов.
Connect to Database	—	Вызвать форму для подключения базы данных в проект.
Connect to Server	—	Вызвать форму для установления связи с сервером.

Название команды	«Горячие» клавиши	Назначение команды
Customize Toolbox	—	Вызвать форму для настройки содержимого Toolbox.
External Tools	—	Вызвать форму для настройки инструментов сторонних разработчиков.
Build Comment Web Pages	—	Построить файл описания проектов на основании XMT-комментариев.
Customize	—	Вызвать форму для настройки содержимого Toolbar.
Options	—	Вызвать форму для настройки основных параметров среды разработки Visual Studio .NET.
Macros	—	Работать с макросами проекта.
Run TemporaryMacro	Ctrl+Shift+P	Запутить макрос проекта на выполнение.
Record TemporaryMacro	Ctrl+Shift+R	Начать запись макроса проекта.
Save TemporaryMacro		Сохранить макрос проекта.
Cancel Recording		Остановить запись макроса.
Macro Explorer	Alt+"	Открыть панель с обзором макросов.
Load Macro Project		Загрузить проект макроса.
Unload Macro Project		Выгрузить проект макроса.
New Macro Project		Создать новый проект макросов.
New Macro Module		Добавить новый модуль в макрос.
New Macro Command		Добавить новую команду в тело макроса.
Run Macro		Запустить редактируемый макрос.
Edit		Открыть редактор тела макроса.

3. СОЗДАНИЕ ПЕРВОГО ПРИЛОЖЕНИЯ

Поскольку наша книга посвящена созданию приложений для Windows, давайте определимся с основными понятиями.

Windows Forms ПРИЛОЖЕНИЕ

Что такое форма

Форма представляет собой экранный объект, обычно прямоугольной формы, который можно применять для предоставления информации пользователю и для обработки ввода информации от пользователя. Формы могут иметь вид стандартного диалогового окна, многодокументного интерфейса (MDI) или поверхности для отображения графической информации. Самый простой способ задать интерфейс пользователя для формы — разместить элементы управления на ее поверхности.

Форма — это объект, который задается свойствами, определяющими их внешний вид, методами, определяющими их поведение, и событиями, определяющими их взаимодействие с пользователем.

Windows Forms в технологии .NET

Формы, как и все объекты в .NET, являются экземплярами классов, унаследованных от System.Windows.Forms.Form. Форма, которую вы создаете с помощью Visual Studio Designer, является классом. Когда вы будете отображать форму во время выполнения программы, этот класс будет использоваться как шаблон для отображения окна.

Необходимо заметить, что Windows Forms предоставляют очень простые и в то же время мощные механизмы для управления графическим интерфейсом пользователя. Если вы измените какое-нибудь свойство, отвечающее за отображение формы на экране, форма сама обновится. Такой дополнительный уровень абстракции позволяет разработчику концентрироваться на его задаче, не заботясь о мелких деталях.

Форму можно создавать полностью в коде программы, однако проще использовать для этого Visual Studio Designer.

Подготовительные операции

Создайте у себя на жестком диске специальную папку, куда сможете сохранять примеры, приведенные в этой книге. Некоторые из них будут

использоваться повторно, поэтому желательно указывать в программах те же имена, что приводятся в книге.

Создание нового проекта

Как создать новый проект с использованием среды Visual Studio .NET, вы уже узнали из главы 2. Не будем повторяться в описании подробностей.

Теперь вам осталось только запустить Visual Studio .NET и выбрать в меню File/New/Project... В появившемся окне выберите Visual C# Project и Windows Application. Назовите проект именем «HelloWorld» и сохраните его в выбранную самостоятельно папку.

Файлы проекта

У вас на экране появится пустая форма. Это стандартный шаблон новой программы Windows Forms. В окне Solution Explorer в ветке проекта «HelloWorld» присутствует четыре элемента: References, App.ico, AssemblyInfo.cs и Forml.cs. Следует заметить, что файлы C# имеют расширение «.cs».

Для вас пока представляет интерес только один файл — Form1.cs. Он содержит код, описывающий вашу форму. Для начала давайте придадим проекту приличный вид: переименуйте файл Form1.cs в MainForm.cs, использовав пункт Rename контекстного меню формы в Solution Explorer. Теперь переименуйте название самой формы. Для этого в окне Properties для вашей формы измените свойство Name с «Form1» на «MainForm»[1].

Свойства проекта

Теперь давайте разберемся с настройками проекта. Каждый проект имеет определенный набор свойств. Среда Visual Studio .NET позволяет изменять эти настройки визуально. Выделите в дереве Solution Explorer корневой элемент HelloWorld. Нажмите пункт меню View/Property Pages. Перед вами появится окно, изображенное на рис. 3.1.

Закладка *Common Properties/General* содержит основную информацию о проекте.

Assembly Name — имя сборки.

Output Type — тип приложения. Здесь можно выбрать Windows Application, Console Application или Class Library. По умолчанию для Windows Forms устанавливается тип Windows Application.

Default Namespace — используемое по умолчанию в проекте пространство имен.

Startup Object — имя класса, содержащего метод Main, который будет вызываться при запуске приложения (подробнее о методе Main читайте в главе *9)*.

---

[1] Если окно Properties отсутствует, в Visual Studio .NET вы можете открыть его, используя меню View/Properties Window или «горячей» клавиши **Alt+Enter**.

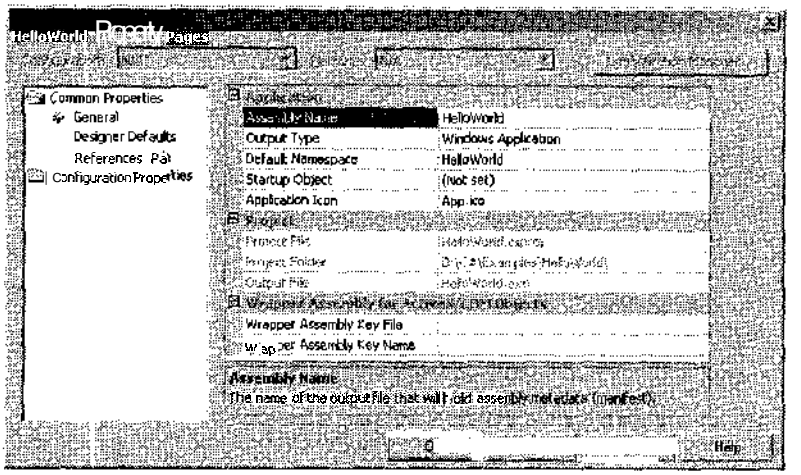


Рис. 3.1. Окно свойств проекта «Common Properties/General»

Application Icon — путь к файлу с пиктограммой для приложения.

Project File — имя файла, содержащего информацию о проекте.

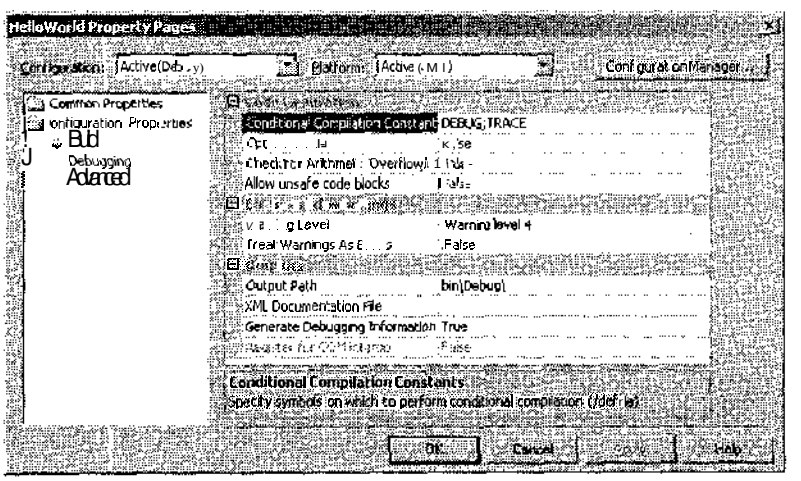
Project Folder — путь к папке, содержащей файл проекта.

Output File — имя выходного файла. Файл с таким именем будет формироваться при построении вашего приложения.

Wrapper Assembly Key File и *Wrapper Assembly Key Name.* Эти свойства проекта выходят за рамки данной книги, поэтому подробнее узнать о применении этих свойств вы сможете, обратившись к Microsoft .NET Framework SDK Documentation, входящей в состав Visual Studio .NET.

Кроме того, вам необходимо знать о свойствах на закладке *Configuration Properties/Build*(рис. 3.2).

Рис. 3.2. Окно свойств проекта «Configuration Properties/Build»



Conditional Compilation Constants —определенные во время компиляции проекта константы. Они помогают разработчику управлять ходом компилирования проекта.

Optimize code. Включение этого свойства в true помогает, в некоторых случаях, увеличить производительность вашей программы в десятки раз.

Check for Arithmetic Overflow/Underflow — контролировать выход результата за границы допустимых значений.

Allow unsafe code blocks —разрешить использование в проекте ключевого слова unsafe.

Warning Level —определить уровень предупреждений, отображаемых при компиляции программы.

Treat Warnings As Errors—воспринимать все предупреждения как ошибки.

Output Path — путь, где будет сформирован выходной файл.

XML Documentation File —имя файла, в который будет записываться документация из комментариев программы. Для формирования документации необходимо использовать меню *Tools/Build Comment Web Pages.* Подробнее об XML-документации читайте в главе 17.

Generate Debugging Information —генерировать отладочную информацию. Эта опция должна быть включена при отладке приложения.

Мы не будем изменять свойства программы. Оставим все значения по умолчанию.

Дизайнер форм

То, что вы видите на экране при создании нового приложения, называется окном дизайнера (рис. 2.3). В этом окне, по сути, в графическом виде отображается код вашей программы. Дизайнер предназначен для удобного и интуитивного создания пользовательского интерфейса программы. К основным элементам дизайнера форм можно причислить:

- Properties Window (пункт меню *View /Properties Window);*
- Layout Toolbar (пункт меню *View/Toolbars/Layout*);
- Toolbox (пункт меню *View/Toolbox*).

Окно кода программы

Как уже отмечалось, в окне дизайнера форм отображается только графическое представление визуальных компонент формы. Все данные вашей программы хранятся кодом программы на языке C#. Для того чтобы посмотреть код созданной нами формы, выберите в контекстном меню элемента MainForm окна Solution Explorer пункт *View Code.* Откроется файл MainForm.cs. Это и есть вся ваша программа, вернее, ее представление синтаксисом языка C#. То же самое вы могли написать в текстовом файле, используя, например, редактор Notepad. Давайте посмотрим, что же представляет собой каркас приложения, созданного Visual Studio .NET.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace HelloWorld
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class MainForm: System.Windows.Forms.Form
  {
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public MainForm()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      iff disposing )
      {
        if (components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }
```

```
    ftregion Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
      //
      // MainForm
      //
      this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
      this.ClientSize = new System.Drawing.Size(292, 273);
      this.Name = "MainForm";
      this.Text = "Form1";

    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
  }
}
```

Для вас, наверное, пока этот текст — китайская грамота. Ничего страшного! Если хватит терпения дочитать книгу до конца, этот код на языке C# будет понятен вам с первого взгляда.

Пока лишь кратко поясним некоторые блоки представленного выше кода. Более подробно все языковые конструкции будут рассмотрены в следующих главах. Цель этой главы — объяснить вам, как просто создать полноценное Windows-приложение на языке C# в среде Visual Studio .NET.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

Этот код определяет, какие пространства имен будут использоваться в данном файле. Каждая представленная строка состоит из двух частей — ключевого слова Using и определяемого пространства имен.

Далее следует объявление собственного пространства имен.

```
namespace HelloWorld
```

Здесь мы объявили собственное пространство имен под названием «HelloWorld». Теперь, если кто-то будет использовать созданные нами в этом пространстве имен элементы, ему придется использовать полное имя объекта: HelloWorld.MainForm.

В нашем пространстве имен объявлен класс формы с именем «MainForm». Внутри фигурных скобок представлена реализация этого класса.

Класс MainForm реализует метод Main. Найдите внутри фигурных скобок следующий код:

```
static void Main()
    {
      Application.Run(new Form1());
    }
```

Функция Main задает точку входа программы, то место, откуда начнется выполнение описанных вами методов (подробнее о функции Main читайте в главе *9)*.

Компиляция программы

То, что у нас написано в файле MainForm.cs, не является исполняемой программой. Это лишь правила, которые определяют, что должен сделать компилятор при создании нашей программы. Для того, чтобы откомпилировать наш код, выберите в меню Build/Build HelloWorld. Внизу среды разработки появится окно — Task List (список задач). В нем будет представлено сообщение:

D:\C#\Examples\HelloWorld\MainForm.cs(72): The type or namespace name 'Form1' could not be found (are you missing a using directive or an assembly reference?)

Это значит, что ваша программа содержит ошибки и не откомпилируется, пока вы их не исправите. Иначе говоря, компилятор не может определить, что такое «Form1», и в скобках дается пояснение (are you missing a using directive or an assembly reference?—«Вы пропускаете директиву using или связку со сборкой?»), которое может помочь устранить проблему (только не в нашем случае).

Output Window

Visual Studio .NET имеет специальное окно, помогающее программистам следить за ходом построения проекта. Открыть это окно можно через пункт меню *View/Other Windows/Output*. Вы можете увидеть в нем ход процесса построения вашей программы. Для нашего проекта результат компиляции представлен следующим образом:

```
Build complete - 1 errors, 0 warnings
```

В результате компиляции обнаружена одна ошибка.

```
---------- Done ----------
Build: 0 succeeded, 1 failed, 0 skipped
```

Построение проекта закончилось сбоем.

Исправление ошибок

Что же мешает нам построить программу? Все очень просто. После создания нового проекта, как вы помните, мы переименовали нашу форму из Form1 в MainForm. Разработчики Visual Studio .NET, конечно, постарались, и код программы автоматически исправил имя класса с Forml на MainForm. Однако среда разработки не такая умная, как нам хотелось бы, поскольку имя было исправлено только в объявлении класса:

```
public class MainForm: System.Windows.Forms.Form
```

А вот использование класса со старого имени «Forml» осталось не измененным.

```
Application.Run(new Form1());
```

Исправьте вышеприведенную строку на строку следующего содержания:

```
Application.Run(new MainForm());
```

Теперь снова попробуйте откомпилировать проект. На этот раз Output Window будет содержать следующее сообщение:

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

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

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

Это означает, что проект успешно построен. Теперь вы можете запустить его.

Запуск приложения

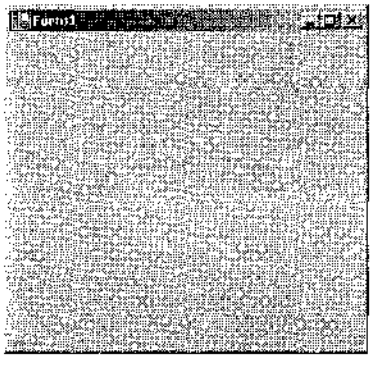


Рис. 3.3. Окно первого приложения

Запуск приложения из среды Visual Studio .NET возможен в двух режимах: с отладкой или без. Для запуска приложения в режиме отладки необходимо выбрать пункт меню *Debug/Start*. Этот режим пригодится вам для отладки создаваемого приложения в ходе выполнения программы. Если же вы хотите запустить приложение только для того, чтобы посмотреть результаты выполненной работы, воспользуйтесь пунктом меню *Debug/Start Without Debugging*.

Запустите приложение в режиме «без отладки» *{Debug/Start Without Debugging)*. На экране появится окно вашего первого приложения (рис. 3.3).

В результате у вас должно было получиться полнофункциональное приложение для Windows со всеми присущими ему атрибутами. Правда, вряд ли какую-то пользу можно извлечь из пустой формы.

Расширение функциональности программы

Давайте расширим наше приложение, добавив к нему кнопку. Для этого переключитесь в окно дизайнера и перетяните из панели Toolbox элемент Button на вашу форму. Поместите эту кнопку по центру формы. Теперь необходимо наделить ее функциональностью. Для этого измените некоторые свойства кнопки в окне *Properties:*

Name: HelloWorld;

Text: Нажми меня.

Теперь необходимо обработать нажатие кнопки. Для этого в окне Properties переключитесь на закладку Events (глава 2). Событие Click предназначено для обработки нажатия кнопки. Щелкните два раза левой кнопкой мыши по ячейке *Click*. Visual Studio .NET переключит вас в окно кода программы. В код были добавлены следующие строки:

```
private void HelloWorld_Click(object sender, System.EventArgs e)
1


}
```

Добавьте в код следующую строку:

```
private void HelloWorld_Click(object sender, System.EventArgs e)
{
  MessageBox.Show("Здравствуй Мир!");
i
```

Рис. 3.4. Сообщение «Здравствуй Мир!»

Откомпилируйте и снова запустите программу. На этот раз приложение содержит добавленную вами кнопку *«Нажми меня»*. Кликните на ней мышью. На экране появится сообщение, изображенное на рис. 3.4.

Как видите, создавать приложения на C# с использованием Visual Studio .NET очень просто.

РАБОТА С КОНСОЛЬЮ

При рассмотрении некоторых глав данной книги в качестве примеров удобнее использовать консольные приложения. Код консольного приложения немного короче, чем приложения Windows Forms. Это избавит нас от длительного процесса создания GUI части программы и позволит сконцентрироваться на рассмотрении функциональной части приложений.

Для работы с консолью в .NET используется класс Console. Преимущества этого класса заключаются в двух аспектах: все его методы являются статическими, так что не нужно создавать для использования его экземпляр. Он объединяет в себе ввод, вывод и вывод ошибок. По умолчанию ввод/вывод производится на стандартную консоль (если ее нет, напри-

мер, в оконных приложениях, вывод просто не осуществляется), но устройства ввода и вывода можно изменить.

Для работы с консолью обычно используются четыре метода: Read, ReadLine, Write и WriteLine, из них первых два — для ввода, последние — для вывода.

Метод Read

Метод Read читает символ из потока ввода. Он возвращает значение типа int, равное коду прочитанного символа, либо -1(минус один), если ничего прочитано не было. Приведем пример программы:

```
do
!
 int i = Console.Read();
 if (i != -1)
 Console.WriteLine("{0} ({1}) ", (char)i, i);
 else
 break;
} while (true);
```

Эта программа показывает на экране введенные символы и их коды.

Метод ReadLine

Метод ReadLine читает из потока ввода строку текста (она завершается символом перевода строки или возврата каретки). Метод возвращает объект типа string или null, если ввод осуществить не удалось.

```
do
{
 string s = Console.ReadLine();
 if (s != null)
 Console.WriteLine("Введенная строка: " + s);
 else
 break;
} while (true);
```

Методы Write и WriteLine

Метод Write выводит на экран значение переданной ему переменной. Он определен для всех базовых типов и поддерживает форматированные строки. Таким образом, можно либо вызвать Write с указанным значением в качестве параметра:

```
Console.Write(1);
Console.Write(0.754);
Console.Write("Hello!"),
```

либо передать строку форматирования и список значений. В строке форматирования применяется множество модификаторов. Здесь мы отметим

лишь то, что вместо «{n}» подставляется n-й входной параметр (нумерация начинается с 0):

```
Console.Write("Привет, {0}", Name);
```

Метод WriteLine отличается от Write только тем, что выводит символ перевода строки в конце.

Я приведу пример создания консольного приложения с использованием наиболее употребляемых операций и дам к ним комментарии. Давайте напишем программу, которая будет осуществлять ввод данных от пользователя, обрабатывать их и выводить на экран.

Откройте окно создания нового проекта, выбрав меню *File/New project.* В списке языков выберите C#, в списке шаблонов — Console Application. Укажите имя для вашего приложения «TestConsole». Откроется окно кода программы на C#. Код программы представлен ниже.

```
using System;
namespace TestConsole
{
  ///<summary>
  /// Summary description for Class1.
  ///</summary>
  class Class1
  {
    ///<summary>
    /// The main entry point for the application.
    ///</summary>
    [STAThread]
    static void Main(string[] args)
    {
      //
      // TODO: Add code to start application here
      //
    }
  }
}
```

Эта программа ничего пока не делает, но она рабочая и готова к запуску. Модифицируйте ваше приложение представленным ниже кодом.

```
using System;
namespace TestConsole
{
  ///<summary>
  /// Summary description for Class1.
  ///</summary>
  class Class1
  {
    ///<summary>
    /// The main entry point for the application.
    ///</summary>
```

```
    [STAThread]
    static void Main(string[] args)
    {
      //объявляем переменную для хранения строки введенных данных
      string strText;

      //выводим на экран информационное сообщение
      Console.WriteLine("Введите Ваше имя.");

 //вводим данные с консоли
      strText = Console.ReadLine();

      //Выводим на экран обработанные данные
      Console.WriteLine("Здравствуйте {0}", strText);
    }
  }
}
```

Раздел II. Фундаментальные понятия

- Основы синтаксиса C#
- Типы данных C#
- Выражения, инструкции и разделители
- Ветвление программ
- Циклические операторы
- Классы
- Методы
- Свойства
- Массивы
- Индексаторы
- Атрибуты
- Интерфейсы
- Делегаты и обработчики событий
- * Особые возможности C# и Visual Studio .NET
- Работа со строками

4. ОСНОВЫ СИНТАКСИСА C#

АЛФАВИТ C#

Алфавит (или множество литер) языка программирования C# составляют символы таблицы кодов ASCII. Алфавит C# включает:

- строчные и прописные буквы латинского алфавита (мы их будем называть буквами);
- цифры от 0 до 9 (назовем их буквами-цифрами);
- символ «_» (подчеркивание — также считается буквой);
- набор специальных символов: " { }, 1 [ ] + — %/ \; ': ? < > = ! & # ~ л. *·
- прочие символы.

Алфавит C# служит для построения слов, которые в C++ называются лексемами. Различают пять типов лексем:

- идентификаторы;
- ключевые слова;
- знаки (символы) операций;
- литералы;
- разделители.

Почти все типы лексем (кроме ключевых слов и идентификаторов) имеют собственные правила словообразования, включая собственные подмножества алфавита.

Лексемы обособляются разделителями. Этой же цели служит множество пробельных символов, к числу которых относятся пробел, табуляция, символ новой строки и комментарии.

ПРАВИЛА ОБРАЗОВАНИЯ ИДЕНТИФИКАТОРОВ

Рассмотрим правила построения идентификаторов из букв алфавита.

- Первым символом идентификатора C# может быть только буква.
- Следующими символами идентификатора могут быть буквы, цифры и нижнее подчеркивание.
- Длина идентификатора не ограничена.

Вопреки правилам словообразования, в C# существуют ограничения относительно применения подчеркивания в качестве самой первой буквы в идентификаторах. Из-за особенностей реализации использование идентификаторов, которые начинаются с этого символа, нежелательно.

РЕКОМЕНДАЦИИ ПО НАИМЕНОВАНИЮ ОБЪЕКТОВ

Имена — это идентификаторы. Любая случайным образом составленная последовательность букв, цифр и знаков подчеркивания с точки зрения грамматики языка идеально подходит на роль имени любого объекта, если только начинается с буквы. Фрагмент программы, содержащий подобную переменную, будет синтаксически безупречен.

И все же имеет смысл воспользоваться дополнительной возможностью облегчить восприятие и понимание последовательностей операторов. Для этого достаточно закодировать с помощью имен содержательную информацию.

Желательно создавать составные осмысленные имена. В этом случае в одно слово можно «втиснуть» предложение, которое в доступной форме представит информацию о типе объекта, его назначении и особенностях использования.

КЛЮЧЕВЫЕ СЛОВА И ИМЕНА

Часть идентификаторов C# входит в фиксированный словарь ключевых слов. Эти идентификаторы образуют подмножество ключевых слов (они так и называются ключевыми словами). Прочие идентификаторы после специального объявления становятся именами. Имена служат для обозначения переменных, типов данных, функций. Обо всем этом позже.

Ниже приводится таблица со списком ключевых слов. Вы не можете использовать эти имена для образования классов, функций, переменных и других языковых структур.

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

КОММЕНТАРИИ

Часто бывает полезно вставлять в программу текст, который является комментарием только для читающего программу человека и игнорируется компилятором. В C# это можно сделать одним из двух способов.

Символы /* начинают комментарий, заканчивающийся символами */ . Такая последовательность символов эквивалентна символу пропуска (например, символу пробела). Это особенно полезно для многострочных комментариев и изъятия частей программы при редактировании, однако следует помнить, что комментарии /* */ не могут быть вложенными.

Символы // начинают комментарий, заканчивающийся в конце строки, на которой они появились. И здесь вся последовательность символов эквивалентна пропуску. Этот способ наиболее полезен для коротких комментариев. Символы /У можно использовать для того, чтобы закомментировать символы /* или */, а символами /* можно закомментировать //.

ЛИТЕРАЛЫ

В C# существует четыре типа литералов:

- целочисленный литерал;
- вещественный литерал;
- символьный литерал;
- строковый литерал.

Литералы — это особая категория слов языка. Для каждого подмножества литералов используются собственные правила словообразования. Мы не будем приводить их здесь, ограничившись лишь общим описанием структуры и назначения каждого подмножества литералов. После этого правила станут более-менее понятны.

Целочисленный литерал служит для записи целочисленных значений и является соответствующей последовательностью цифр (возможно, со знаком '-'). Целочисленный литерал, начинающийся со знака 0, воспринимается как восьмеричное целое. В этом случае цифры 8 и 9 не должны встречаться среди составляющих литерал символов. Целочисленный литерал, начинающийся с Ох или ОХ, воспринимается как шестнадцатеричное целое. В этом случае целочисленный литерал может включать символы от A или a, до F или f, которые в шестнадцатеричной системе эквивалентны десятичным значениям от 10 до 15. Непосредственно за литералом могут располагаться в произвольном сочетании один или два специальных суффикса: U (или и) и L (или l).

Вещественный литерал служит для отображения вещественных значений. Он фиксирует запись соответствующего значения в обычной десятичной или научной нотациях. В научной нотации мантисса отделяется от порядка литерой E (ли e). Непосредственно за литералом может располагаться один из двух специальных суффиксов: F (или f) и L (или l).

Значением *символьного литерала* является соответствующее значение ASCII кода (это, разумеется, не только буквы, буквы-цифры или специальные символы алфавита C#). Символьный литерал представляет собой последовательность одной или нескольких литер, заключенных в одинарные кавычки. Символьный литерал служит для представления литер в одном из форматов представления. Например, литера Z может быть представлена литералом «Z», а также литералами «\132» и «\x5A». Любая литера может быть представлена в нескольких форматах представления: обычном, восьмеричном и шестнадцатеричном.

Строковые литералы являются последовательностью (возможно, пустой) литер в одном из возможных форматов представления, заключенных в двойные кавычки. Строковые литералы, расположенные последовательно, соединяются в один литерал, причем литеры соединенных строк остаются различными. Так, последовательность строковых литералов «\xF» «F» после объединения будет содержать две литеры, первая из которых является символьным литералом в шестнадцатеричном формате «\xF», вторая — символьным литералом «F». Строковый литерал и объединенная последовательность строковых литералов заканчиваются пустой литерой, которая используется как индикатор конца литерала.

5. ТИПЫ ДАННЫХ C#

C# является жестко типизированным языком. При его использовании вы должны объявлять тип каждого объекта, который создаете (например, целые числа, числа с плавающей точкой, строки, окна, кнопки, и т. д.), и компилятор поможет вам избежать ошибок, связанных с присвоением переменным значений только того типа, который им соответствует. Тип объекта указывает компилятору размер объекта (например, объект типа int занимает в памяти 4 байта) и его свойства (например, форма может быть видима и невидима, и т.д.).

Подобно языкам C++ и Java, C# подразделяет типы на два вида: встроенные типы, которые определены в языке, и определяемые пользователем типы, которые выбирает программист.

C# также подразделяет типы на две другие категории: размерные и ссылочные. Основное различие между ними — это способ, которым их значения сохраняются в памяти. Размерные типы сохраняют свое фактическое значение в стеке. Ссылочные типы хранят в стеке лишь адрес объекта, а сам объект сохраняется в куче. Куча — основная память программ, доступ к которой осуществляется на много медленнее чем к стеку. Если вы работаете с очень большими объектами, то сохранение их в куче имеет много преимуществ.

В главе 5 будут подробно рассмотрены различные преимущества и недостатки работы с ссылочными типами.

C# также поддерживает и указатели на типы, но они редко употребляются. Применение указателей связано с использованием неуправляемого кода.

ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ СТЕКА И КУЧИ

Стек — это структура данных, которая сохраняет элементы по принципу: первым пришел, последним ушел (полная противоположность очереди). Стек относится к области памяти, поддерживаемой процессором, в которой сохраняются локальные переменные. Доступ к стеку во много раз быстрее, чем к общей области памяти, поэтому использование стека для хранения данных ускоряет работу вашей программы. В C# размерные типы (например, целые числа) располагаются в стеке: для их значений зарезервирована область в стеке, и доступ к ней осуществляется по названию переменной.

Ссылочные типы (например, объекты) располагаются в куче. Куча — это оперативная память вашего компьютера. Доступ к ней осуществляется медленнее, чем к стеку. Когда объект располагается в куче, то переменная хранит лишь адрес объекта. Этот адрес хранится в стеке. По

адресу программа имеет доступ к самому объекту, все данные которого сохраняются в общем куске памяти (куче).

«Сборщик мусора» уничтожает объекты, располагающиеся в стеке, каждый раз, когда соответствующая переменная выходит за область видимости. Таким образом, если вы объявляете локальную переменную в пределах функции, то объект будет помечен как объект для «сборки мусора». И он будет удален из памяти после завершения работы функции.

Объекты в куче тоже очищаются сборщиком мусора, после того как конечная ссылка на них будет разрушена.

ВСТРОЕННЫЕ ТИПЫ

Язык C# предоставляет программисту широкий спектр встроенных типов, которые соответствуют CLS (Common Language Specification) и отображаются на основные типы платформы .NET. Это гарантирует, что объекты, созданные на C#, могут успешно использоваться наряду с объектами, созданными на любом другом языке программирования, поддерживающем .NET CLS (например, VB.NET).

Каждый тип имеет строго заданный для него размер, который не может изменяться. В отличие от языка C++, в C# тип int всегда занимает 4 байта, потому что отображается к Int32 в .NET CLS. Представленная ниже таблица содержит список всех встроенных типов, предлагаемых C#.

Тип	Область значений	Размер
sbyte	-128 ДО 127	Знаковое 8-бит целое
byte	0 до 255	Беззнаковое 8-бит целое
char	U + 0000 до U + ffff	16-битовый символ Unicode
bool	true или false.	1 байт
short	-32,768 до 32,767	Знаковое 16-бит целое
ushort	0 до 65,535	Беззнаковое 16-бит целое
int	-2,147,483,648 до 2,147,483,647	Знаковое 32-бит целое
uint	0 ДО 4,294,967,295	Беззнаковое 32-бит целое
long	-9,223,372,036,854,775,808 ДО 9,223,372,036,854,775,807	Знаковое 32-бит целое
ulong	0 до 18,446,744,073,709,551,615	Беззнаковое 32-бит целое
float	$\pm 1.5 \cdot 10^{-45}$ до $\pm 3.4 \cdot 10^{38}$	4 байта, точность — 7 разрядов
double	$\pm 5.0 \cdot 10^{-324}$ до $\pm 1.7 \cdot 10^{308}$	8 байт, точность — 16 разрядов
decimal		12 байт, точность —28 разрядов

В дополнение к этим примитивным типам C# может иметь объекты типа enum и struct (см. ниже).

Преобразование встроенных типов

Объекты одного типа могут быть преобразованы в объекты другого типа неявно или явно. Неявные преобразования происходят автоматически, компилятор делает это вместо вас. Явные преобразования осуществляются, когда вы «приводите» значение к другому типу. Неявные преобразования гарантируют также, что данные не будут потеряны. Например, вы можете неявно приводить от short (2 байта) к int (4 байта). Независимо от того, какой значение находится в short, оно не потеряется при преобразовании к int:

```
short x = 1;
int y = x;//неявное преобразование
```

Если вы делаете обратное преобразование, то, конечно же, можете потерять информацию. Если значение в int больше, чем 32.767, оно будет усечено при преобразовании. Компилятор не станет выполнять неявное преобразование от int к short:

```
short x;
int y = 5;
x = y;  //не скомпилируется
```

Вы должны выполнить явное преобразование, используя оператор приведения:

```
short x;
int y = 5;
x = (short) y;//OK
```

ПЕРЕМЕННЫЕ

Переменная — это расположение в памяти объекта определенного типа. В приведенных выше примерах x и y — переменные. Переменные могут иметь значения, которыми они проинициализированы, или эти значения могут быть изменены программно.

Назначение значений переменным

Чтобы создать переменную, вы должны задать тип переменной и затем дать этому типу имя. Вы можете проинициализировать переменную во время ее объявления или присвоить ей новое значение во время вьшолнения программы. Вот пример программы, который в первом случае использует инициализацию для присвоения значения переменной, во втором случае использует присвоение значения переменной с помощью оператора «=»:

```
class Variables
{
 static void Main()
 {
 int myInt = 10;
```

```
System.Console.WriteLine("Инициализированная переменная myInt: {0}",
myInt);
myInt = 5;
System.Console.WriteLine("myInt после присвоения значения: {0}",
myInt);
}
}
```

Результат работы этой программы будет следующий:

Инициализированная переменная myInt: 10
myInt после присвоения значения: 5

Здесь строка:

```
int myInt = 10;
```

означает объявление и инициализацию переменной mylnt. Строка:

```
myInt = 5;
```

означает присвоение переменной mylnt значения 5.

Определение значений переменных

C# требует определения значений, то есть переменные перед использованием должны быть инициализированы. Чтобы проверить это правило, давайте рассмотрим следующий пример:

```
class Variables
{
 static void Main()
 {
 int myInt;
 System.Console.WriteLine("Initialized, mylnt: {0}",
 myInt);
 mylnt = 5;
 System.Console.WriteLine("After assignment, myInt: {0}",
 myInt);
 }
}
```

Если вы попробуете скомпилировать этот пример, компилятор отобразит следующее сообщение об ошибке:

```
error CS0165: Use of unassigned local variable 'myInt'
```

Нельзя использовать неинициализированную переменную в C#. У вас может сложиться впечатление, что вы должны инициализировать каждую переменную в программе. Это не так. Вы должны лишь назначить переменной значение прежде, чем попытаетесь ее использовать. Ниже представлен пример правильной программы без использования инициализации переменной mylnt.

```
class Variables
{
 static void Main()
```

```
{
int mylnt;
mylnt: - 1C;
System.Console.WriteLine("Initialized, mylnt: {0}",
myInt);
mylnt - 5;
System.Console.WriteLine("After assignment, myInt: {0}",
mylnt];
i
}
```

В данном примере вместо инициализации выбирается присвоение значения переменной mylnt до ее использования:

```
mylnt = 10;
```

КОНСТАНТЫ

Константа — это переменная, значение которой не может быть изменено. Переменные — это более гибкий способ хранения данных. Однако иногда вы хотите гарантировать сохранение значения определенной переменной. Например, число pi. Как известно, значение этого числа никогда не изменяется. Следовательно, вы должны гарантировать, что переменная, хранящая это число, не изменит своего значения на протяжении всей работы программы. Ваша программа будет лучше читаемой, если вы вместо записи:

```
y - x * 3.14159265358979323846264338327 95
```

будете использовать переменную, которая хранит значение pi. В таком случае используемой переменной должна быть константа:

```
const double pi - 3.14159265358979323846264338327 95;
y = x * pi;
```

Существуст три разновидности константы: литералы, символические константы и перечисления. Рассмотрим следующий случай:

```
x = 100;
```

Значение 100 — это литеральная константа. Значение 100 — это всегда 100. Вы не можете установить новое значение на 100. Вы не можете сделать так, чтобы 100 представляло значение 99. Символические константы устанавливают имя для некоторого постоянного значения. Вы объявляете символическую константу, используя ключевое слово const, и применяете следующий синтаксис для создания константы:

```
const тип идентификатор = значение;
```

Константа обязательно должна быть проинициализирована, и ее значение не может быть изменено во время выполнения программы. Например:

```
const double pi = 3.14159265358979323846264338327 95;
pi - 5.0; //недопустимая операция
```

В этом примере число 3.14... — литеральная константа, а pi — символическая константа типа double. Ниже представлен пример использования символических констант.

```
class Constants
{
 static void Main()
 {
 const in: pi = 3.14159265358979323846264338327 95;
 const int g = 9.81; //гравитационная постоянная

 System.Console.WriteLine("Число пи: {0}",
 pi);
 System.Console.WriteLine("Гравитационная постоянная: {0}",
 g);
 }
}
```

Результат работы программы будет следующий:

Числоpi:3.14159265358979323846264338327 95

Гравитационная постоянная: 9.81

В последнем примере создаются две символические целочисленные константы: pi и д. Эти константы преследуют ту же цель, что и использование их литеральных значений. Но данные константы имеют имена, которые несут в себе гораздо больше информации об используемом значении, чем просто набор цифр.

Для того чтобы убедиться, что константные значения не могут быть изменены во время выполнения программы, добавьте в код следующую строку:

```
д = 10;
```

Во время компиляции вы получите следующее сообщение об ошибке:

error CS0131: The left-hand side of an assignment must be a variable, property or indexer.

ПЕРЕЧИСЛЕНИЯ

Перечисления являются мощной альтернативой константам. Это особый тип значений, который состоит из набора именованных констант. Допустим, у вас есть список констант, содержащих годы рождения ваших знакомых. Для того чтобы запрограммировать это при помощи констант, вам придется написать:

```
const intfloat maryBirthday = 1955;
const intfloat ivanBirthday = 1980;
const intfloat pavelBirthday = 1976;
```

У вас получились три совершенно несвязанные константы. Для того чтобы установить логическую связь между ними, в C# предусмотрен механизм перечислений. Вот как выглядит тот же код, записанный при помощи перечислений:

```
enum FriendsBirthday
f
const float maryBirthday - 1955;
```

```
const float ivanBirthday = 1980;
const float pavelBirthday = 1976;
}
```

Теперь две символические константы являются элементами одного перечисления типа FriendsBirthday.

Каждое перечисление имеет свой базовый тип, которым может быть любой встроенный целочисленный тип C# (int, long, short и т. д.), за исключением char.

Перечисление задается следующим образом:

```
[атрибуты] [модификаторы] enum идентификатор[: базовый тип] { список перечислений);
```

Атрибуты и модификаторы рассматриваются далее в этой книге. Пока давайте остановимся на второй части этого объявления. Перечисление начинается с ключевого слова enum, которое сопровождается идентификатором типа:

```
enum MyEnumerators
```

Базовый тип — основной тип для перечисления. Если вы не учитываете этот описатель при создании перечисления, то будут использоваться значения по умолчанию int. Но вы можете применить любой из целочисленных типов (например, ushort, long), за исключением char. Например, следующий фрагмент кода объявляет перечисление целых чисел без знака (uint):

```
enum Sizes: uint
{
 Small = 1,
 Middle= 2,
 Large = 3
}
```

Внутри каждого перечисления записывается список возможных значений перечисления, разделенных запятой. Каждое значение может представлять собой либо просто набор символических констант, либо набор символических констант в сочетании с литеральным целочисленным значением. Если вы не укажете для элементов перечисления целочисленных значений, то компилятор пронумерует их сам, начиная с 0. Например, следующие фрагменты кода аналогичны:

```
enum Sizes
 (
 Small = 0,
 Middle= 1,
 Large = 2
 }
И
enum Sizes
{
 Small,
 Middle,
 Large
}
```

Если вы объявите свое перечисление следующим образом:

```
enum Sizes
[
 Small,
 Middle-20,
 Large
'i
```

то элементы перечисления будут иметь такие числовые значения:

Small = 0;
Middle = 20;
Large = 21.

Давайте рассмотрим пример, который наглядно показывает, каким образом перечисления помогают упростить код приложения.

```
class ScreenResolutions
{
  //перечисление размеров мониторов в дюймах
  enum Screens
  (
    Small = 14,
    Medium = 17,
    Large - 19,
    SuperLarge -- 21,
  }

  static void Main( }
  {
    System.Console.WriteLine("Самые маленькие мониторы имеют размер: {0}",
      (int) Screens.Small );
    System.Console.WriteLine("Самые большие мониторы имеют размер: {0}",
      (int) Screens.SuperLarge);

  }
}
```

Как видите, значение перечисления (SuperLarge) должно быть специфицировано именем перечисления (Screens). По умолчанию, значение перечисления представляется его символическим именем (типа Small или Medium). Если вы хотите отобразить значение константы перечисления, то должны привести константу к ее основному типу (в данном случае int). Целочисленное значение передается в функцию WriteLine, и оно отображается на экране.

Строковые константы

Для объявления в программе константной строки вам необходимо заключить содержимое строки в двойные кавычки ("My string"). Вы можете делать это практически в любом месте программы: в передаче парамет-

ров функции, в инициализации переменных. Мы уже неоднократно применяли строковые константы при выводе данных на экран.

```
System.Console.WriteLine("Самые большие мониторы имеют размер: {C}",
   (int) Screens.SuperLarge);
```

Здесь в качестве одного из параметров функции используется строка "Самые большие мониторы имеют размер: {0}".

```
string strMessage = "Здравствуй Мир!";
```

В данном случае константная строка «Здравствуй Мир!» инициализирует переменную strMessage.

МАССИВЫ

Более детально работа с массивами будет рассмотрена в главе 12. Пока я лишь хочу рассказать о массивах с точки зрения типов данных.

Массивы в C# несколько отличаются от других C-подобных языков. Начнем сразу с примеров. Пример первый:

```
int [ ] к; //к-массив
k=new int [3]; //Определяем массив из 3 целых
k[0]=-5; к[1]=4; k[2]=55; //Задаем элементы массива
//Выводим третий элемент массива
Console.WriteLine(k[2].ToString());
```

Смысл приведенного фрагмента ясен из комментариев. Обратите внимание на некоторые особенности. Во-первых, массив определяется именно как

```
int[] к;
```

а не как один из следующих вариантов:

```
int k[]; //Неверно!
int k{3} ; //Неверно!
int[3] к; //Неверно!
```

Во-вторых, так как массив представляет собой ссылочный объект, то для создания массива необходима строка

```
k=new int [3];
```

Именно в ней мы и определяем размер массива. Кроме того, возможны конструкции вида

```
int[] k = new int [3];
```

Элементы массива можно задавать сразу при объявлении. Например:

```
int[] к = {-5, 4, 55};
```

Разумеется, приведенные конструкции применимы не только к типу int и не только к массиву размера 3.

В C#, как и в C/C++, нумерация элементов массива идет с нуля. Таким образом, в нашем примере начальный элемент массива — это k[0], а последний — k[2]. Элемента к[3], естественно, нет.

Теперь переходим к многомерным массивам. Вот так задается двумерный массив:

```
intf,|" k = new int [2,3];
```

Обратите внимание, что пара квадратных скобок только одна. В нашем примере у массива 6 (=2*3) элементов (к[0,0] — первый, k[1,2] — последний).

Аналогично мы можем задавать многомерные массивы. Вот пример трехмерного массива:

```
int[,,] к = new int [10,10,10];
```

А вот так можно сразу инициализировать многомерные массивы:

```
int[,] k = {{2,-2},{3,-22},{0,4}};
```

Приведенные выше примеры многомерных массивов называются прямоугольными. Если их представить в виде таблицы (в двумерном случае), то массив будет представлять собой прямоугольник.

Наряду с прямоугольными массивами существуют так называемые ступенчатые. Вот пример:

```
//Объявляем 2-мерный ступенчатый массив
int[][] k = new int [2][];
//Объявляем 0-й элемент нашего ступенчатого массива
//Это опять массив и в нем 3 элемента
k[0]=new int[3];
//Объявляем 1-й элемент нашего ступенчатого массива
//Это опять массив и в нем 4 элемента
k[1]=new int[4];
k[1][3]=22; //записываем 22 в последний элемент массива
```

Обратите внимание, что у ступенчатых массивов мы задаем несколько пар квадратных скобок (по размерности массива). И точно так же мы что-нибудь делаем с элементами массива — записываем, читаем и т. п.

Самая важная и интересная возможность ступенчатых массивов — это их «непрямоугольность». Так, в приведенном выше примере в первой «строке» массива к три целых числа, а во второй — четыре. Часто это оказывается очень кстати.

6. ВЫРАЖЕНИЯ, ИНСТРУКЦИИ И РАЗДЕЛИТЕЛИ

ВЫРАЖЕНИЯ (Expressions)

Выражение — это строка кода, которая определяет значение. Пример простого выражения:

```
myValue = 100;
```

Обратите внимание, что данная инструкция выполняет присвоение значения 100 переменной myValue. Оператор присвоения (=) не сравнивает стоящее справа от него значение (100) и значение переменной, которая находится слева от оператора (myValue). Оператор «=» устанавливает значение переменной myValue равным 100.

Поскольку myValue = 100, то как выражение, которое определяет значение 100, myValue может использоваться другим оператором присвоения. Например:

```
mySecondValue = myValue = 100;
```

В данном выражении литеральное значение 100 присваивается переменной myValue, а затем оператором присвоения устанавливается вторая переменная mySecondValue с тем же значением 100. Таким образом, значение 100 будет присвоено обеим переменным одновременно. Инструкцией такого вида вы можете инициализировать любое число переменных с одним и тем же значением, например 20:

```
a = b = c = d = e = 20;
```

ИНСТРУКЦИИ (Statements)

Инструкция — это законченное выражение в коде программы. Программа на языке C# состоит из последовательностей инструкций. Каждая инструкция обязательно должна заканчиваться точкой с запятой (;). Например:

```
int x;  // инструкция
x = 100;   //другая инструкция
int y = x;  //тоже инструкция
```

Кроме того, в C# существуют составные инструкции. Они, в свою очередь, состоят из набора простых инструкций, помещенных в фигурные скобки ({ }).

```
{
int x;  // инструкция
x = 100;   //другая инструкция
```

```
int y = x;  //тоже инструкция
}
```

В этом примере все три инструкции являются элементами одной инструкции.

C# инструкции рассматриваются в соответствии с порядком их записи в тексте программы. Компилятор начинает рассматривать код программы с первой строки и заканчивает концом файла.

РАЗДЕЛИТЕЛИ (Delemiters)

В языке C# пробелы, знаки табуляции и переход на новую строку рассматриваются как разделители. В инструкциях языка C# лишние разделители игнорируются. Таким образом, вы можете написать:

```
myVaiue = 100;
```

или:

```
myVaiue = 100;
```

Компилятор обработает эти две инструкции как абсолютно идентичные. Исключение состоит в том, что пробелы в пределах строки не игнорируются. Если вы напишете:

```
Console.WriteLine("Здравствуй    !");
```

каждый пробел между словами «Здравствуй», «Мир» и знаком «!» будет обрабатываться как отдельный символ строки.

В большинстве случаев использование пробелов происходит чисто интуитивно. Обычно они применяются для того, чтобы сделать программу более читаемой для программиста; для компилятора разделители абсолютно безразличны.

Надо заметить, что есть случаи, в которых использование пробелов является весьма существенным. Например, выражение:

```
int myVaiue = 25;
```

то же самое, что и выражение:

```
int myValue=25;
```

но следующее выражение не будет соответствовать двум предыдущим:

```
intmyValue =25;
```

Компилятор знает, что пробел с обеих сторон оператора присвоения игнорируется (сколько бы много их не было), но пробел между объявлением типа int и именем переменной myVaiue должен быть обязательно. Это не удивительно, пробелы в тексте программы позволяют компилятору находить и анализировать ключевые слова языка. В данном случае это int, а некоторый термин intmyValue для компилятора неизвестен. Вы можете свободно добавлять столько пробелов, сколько вам нравится, но между int и myVaiue должен быть, по крайней мере, один символ пробела или табуляции.

7. ВЕТВЛЕНИЕ ПРОГРАММ

Для того чтобы программы на C# были более гибкими, используются операторы перехода (операторы ветвления). В C# есть два типа ветвления программы: безусловный переход и условный.

Кроме ветвлений, в C# также предусмотрены возможности циклической обработки данных, которые определяются ключевыми словами: for, while, do, in и foreach. Эти операторы будут обсуждаться в главе «Циклическая обработка данных». Пока давайте рассмотрим некоторые из основных способов условного и безусловного перехода.

БЕЗУСЛОВНЫЕ ПЕРЕХОДЫ

Безусловный переход осуществляется двумя способами.

Первый способ — это вызов функций. Когда компилятор находит в основном тексте программы имя функции, то происходит приостановка выполнения текущего кода программы и осуществляется переход к найденной функции. Когда функция выполнится и завершит свою работу, то произойдет возврат в основной код программы, на ту инструкцию, которая следует за именем функции. Следующий пример иллюстрирует безусловный переход с использованием функции:

```
using System;
class Functions
{
  static void Main( )
  {
    Console.WriteLine("Метод Main. Вызываем метод Jump...");
    Jump( );
    Console.WriteLine ("Возврат в метод Main.");
  }
  static void Jump( )
  {
    Console.WriteLine("Работает метод Jump!");
  }
}
```

Откомпилируйте и запустите программу. Вы увидите результат ее работы:

Метод Main. Вызываем метод Jump...
Работает метод Jump!
Возврат в метод Main.

Программа начинает выполняться с метода Main() и осуществляется последовательно, пока компилятор не вызовет функцию Jump(). Таким образом, происходит ответвление от основного потока выполняемых инструкций. Когда функция Jump() заканчивает свою работу, то продолжается выполнение программы со следующей строки после вызова функции.

Второй способ реализации безусловного перехода можно осуществить при помощи ключевых слов: goto, break, continue, return или throw. Дополнительная информация об инструкциях goto, break, continue и return будет рассмотрена ниже в этой главе. Об инструкции throw мы поговорим в главе «Обработка исключений».

УСЛОВНЫЕ ПЕРЕХОДЫ

Условный переход можно реализовать в программе с помощью ключевых слов языка: if, else или switch. Такой переход возможен только при условии, если он является истинным.

if...else оператор

if...else — это оператор ветвления, работа которого определяется условием. Условие оператора анализируется инструкцией if. Если условие верно (true), то выполняется блок инструкций программы, описанных после условия.

```
if ( expression ) statement 1
 [else statement2 ]
```

Такой вид этого оператора вы можете найти в документации по C#. Он показывает, что работа условного оператора определяется булевым выражением (выражение, которое имеет значение true или false) в круглых скобках. Если значение этого выражения истинно, то выполняется блок инструкций statementl. Если же выражение ложно, произойдет выполнение блока инструкций statement2. Необходимо заметить, что вторая часть оператора (else statement2) может не указываться. Если инструкций в блоках statementl или statement2 больше одной, то блок обязательно нужно брать в фигурные скобки.

```
using System;
class Conditional
{
  static void Main( )
  {
    int valueOne = 20;
    //устанавливаем второе значение больше первого
    int valueTwo = 10;

    if ( valueOne > valueTwo )
      Console.WriteLinef
```

```
        "valueOne: {0} больше чем valueTwo: {1}",
        valueOne, valueTwo);
    else
      Console.WriteLine(
        "valueTwo: {0} больше или равно valueOne: {1}",
        valueTwo, valueOne);

    //устанавливаем первое значение больше второго
    valueTwo = 30;
    if ( valueOne > valueTwo )
    {
      Console.WriteLine(
        "valueOne: {0} больше чем valueTwo: (1)",
        valueOne, valueTwo);
    i

    //делаем значения одинаковыми
    valueOne = valueTwo;

    if(valueOne == valueTwo)
    {
      Console.WriteLine(
        "valueOne и valueTwo равны: {0}=={1}",
        valueOne, valueTwo);
    }
  }
}
```

Операторы сравнения: больше чем (>), меньше чем (<) или равно (==) достаточно понятны и просты в использовании.

В этом примере первый оператор if проверяет значение переменной valueOne относительно значения переменной valueTwo. Если значение valueOne больше значения переменной valueTwo (valueOne будет равным 20, а valueTwo будет равным 10), то условие оператора (valueOne > valueTwo) является истинным и на экране появится строка:

valueOne: 10 больше чем valueTwo: 1

Во втором операторе if условие осталось прежним, но значение valueTwo уже изменилось на 30. Поэтому условие оператора (valueOne > valueTwo) является ложным и результат не будет отображен на экране.

Последний оператор условия проверяет равенство двух значений (valueOne == valueTwo). А поскольку перед этим мы приравняли две переменные, то результат проверки будет истинным. И на экране появится информация:

valueOne и valueTwo равны: 30==30

Результатом работы программы будет следующая информация:

valueTwo: 20 больше или равно valueOne: 10

valueOne и valueTwo равны: 30==30

Вложенные операторы условия

Для обработки сложных условий возможно вложение условных операторов в блоки инструкций других условных операторов. Например, вам необходимо оценить рабочую температуру воздуха в офисе и выдать сообщение пользователю.

Если значение температуры больше 21° и меньше 26°, то температура находится в пределах нормы. Если же температура равна 24°, то выдается сообщение о том, что рабочий климат оптимален.

Есть много способов написать такую программу. Приведем пример, иллюстрирующий один из способов, в котором используется вложенность оператора if...else:

```
using System;
class Values
{
  static void Main( )
  {
    int temp = 25;
    if (temp > 21)
    {
      if (temp < 26)
      {
        Console.WriteLine(
          "Температура находится в пределах нормы");

        if (temp == 24)
        {
          Console.WriteLine("Рабочий климат оптимален");
        )
        else
        {
          Console.WriteLine("Рабочий климат не оптимален\n" +
              "Оптимальная температура 24");
        }
      }
    }
  }
1
```

В теле программы встречается два условных оператора if...else. В первом операторе происходит проверка на попадание значения температуры в нижний предел (21°). Значение temp больше, чем 21, значит условие (temp > 21) истинное и выполнится следующая проверка. Во втором операторе происходит проверка на попадание значения температуры в верхний предел (26°). Значение temp равно 24, значит, условие (temp < 26) истинное и будет выполняться блок инструкций в фигурных скобках. Таким образом, уже выяснилось, что температура в пределах нормы, и

остается узнать, является ли она оптимальной — это и реализует последний оператор if. Значение temp равно 25°, значит, условие (temp == 24) последнего оператора ложно и на экране появится сообщение:

Температура находится в пределах нормы
Рабочий климат не оптимален.
Оптимальная температура 25

Заметьте, что в условном операторе в круглых скобках стоит два знака равно:

```
if(tempValue == 24)
```

Если бы тут стоял один знак, то такую ошибку было бы трудно заметить. И результатом такого выражения стало бы присвоение переменной temp значения 24. В C и C ++ любое значение, отличное от нуля, определяется как булева истина (true), следовательно, условный оператор вернул бы истину, и на экране вывелась бы строка «Рабочий климат не оптимален». Таким образом, действие выполнилось бы неправильно, а побочным эффектом стало бы нежелательное изменение значения temp. Впоследствии у разработчика могли бы возникать непредвиденные ошибки работы программы!

C# требует, чтобы условные операторы принимали в качестве условий только булевы значения. Поэтому, если бы такая ошибка возникла, то компилятор не смог бы преобразовать выражение temp = 24 к булеву типу и выдал бы сообщение об ошибке в процессе компиляции программы. Это является достоинством по сравнению с C++, так как разрешается проблема неявных преобразований типов данных, в частности целых чисел и булева типа!

Использование составных инструкций сравнения

Оператор if в инструкции сравнения может применять несколько инструкций, объединенных арифметическими операторами. В качестве последних используются операторы (&& — И), (| — ИЛИ) и (! — НЕ).

Рассмотрим пример использования составных инструкций в блоке if.

```
using System;

namespace C_Sharprogramming
{
  class Conditions
  {
    static void Main(string[] args)
    {
      int n1 = 5;
      int n2 = 0;
      if((n1 == 5) SS (n2 == 5))
      Console.WriteLine("Инструкция И верка");
      else
      Console.WriteLine("Инструкция и не верна"];
```

```
        if((n1 == 5)  ||  (n2 == 5))
        Console.WriteLine("Инструкция ИЛИ верна");
        else
        Console.WriteLine("Инструкция ИЛИ не верна");
      }
    }
}
```

В данном примере каждая инструкция if проверяет сразу два условия. Первое if условие использует оператор (&&) для проверки условия. Необходимо, чтобы сразу два условия в блоке if были истинны. Только тогда выражение будет считаться верным:

```
if!(ni == 5)  &&  (n2 == 5) )
```

При этом не всегда выполняются все выражения в блоке if. Если первое выражение однозначно определяет результат операции, то второе уже не проверяется. Так, если бы условие nl == 5 было ложным, то условие п2 == 5 уже не проверялось бы, поскольку его результат перестал бы играть роль.

Второе выражение использует оператор (||) для проверки сложного условия:

```
if((nl == 5) | : (n2 == 5))
```

В данном случае для верности всего выражения достаточно истинности лишь одного из условий. И тогда действует правило проверки условий до выяснения однозначности результата. То есть, если условие nl == 5 верно, то п2 == 5 уже проверяться не будет. Это свойство может сыграть очень важную роль при разработке кода программ. Давайте рассмотрим пример.

```
using System;

namespace C_Sharprogramming
{
  class Conditions
  {
    static void Main(string[] args)
    {
      const int MAX_VALUE = 3;
      int nl = 1;
      int n2 = 1;
      if((++nl < MAX_VALUE)  &&  (++n2 < MAX_VALUE))
      Console.WriteLine("Операция && nl:={0} n2:={1}", nl, n2);
      if((++nl < MAX_VALUE+1)  II  (++n2 < MAX_VALUE+1))
      Console.WriteLine("Операция ! | nl:-{0} n2:={1}", nl, n2);
    }
  }
}
```

Результат работы программы будет следующий:

Операция && nl:=2 n2:=2
Операция || nl:=3 n2:=2

Заметьте, что первый оператор if нарастил значения обеих переменных, потому как осуществлялась проверка обоих условий. Второй оператор if выполнил инкремент только переменной n1. Причина в том, что условие (++n1<MAX_VALUE+1) уже однозначно определяет весь блок if как истинный, поэтому дальнейшая проверка не выполняется, как и все действия, связанные с проверкой оставшихся условий.

Оператор switch как альтернатива оператору условия

Достаточно часто встречаются ситуации, когда вложенные условные операторы выполняют множество проверок на совпадение значения переменной, но среди этих условных операторов только один является истинным.

```
if (myValue == 10) Console.WriteLine("myValue равно 10");
else
if (myValue == 20) Console.WriteLine("myValue равно 20 " ) ;
else
if (myValue == 30) Console.WriteLine("myValue равно 30 " ) ;
else ....
```

Когда вы имеете такой сложный набор условий, лучше всего воспользоваться оператором switch, который является более удобной альтернативой оператору if.

Логика оператора switch следующая: «найти значение, соответствующее переменной для сравнения, и выполнить соответствующее действие». Иными словами, он работает как оператор выбора нужного действия.

```
switch (выражение)
{
  case константное выражение: инструкция
  выраженке перехода
  [default: инструкция]
}
```

Вы видите, что, подобно оператору условия if...else, выражение условия помещено в круглые скобки в начале оператора switch.

Внутри оператора switch есть секция выбора — case и секция действия по умолчанию — default. Секция выбора нужна для определения действия, которое будет выполняться при совпадении соответствующего константного выражения выражению в switch. В этой секции обязательно нужно указать одно или несколько действий. Секция default может в операторе switch не указываться. Она выполняется в том случае, если не совпала ни одна константная инструкция из секции выбора.

Оператор case требует обязательного указания значения для сравнения (constant-expression) — константного выражение (литеральная или символическая константа, или перечисление), а также блока инструкций (statement) и оператора прерывания действия (jump-statement).

Если результат условия совпадет с константным значением оператора case, то будет выполняться соответствующий ему блок инструкций. Как

правило, в качестве оператор перехода используют оператор break, который прерывает выполнение оператора switch. Альтернативой ему может быть и другой оператор — goto, который обычно применяют для перехода в другое место программы.

Чтобы вы могли увидеть, как оператор switch заменяет сложный набор условий, приведем пример той же программы, но с использованием оператора switch:

```
switch ( myValue )
{
case 10:
  Console.WriteLine("myValue равно 10" | ;
  break;
case 20:
  Console.WriteLine("myValue равно 20");
  break;
  case 30: Console.WriteLine("myValue равно 30"];
  break;
}
```

В этом примере мы проверяем значение переменной myValue на равенство ее одному из следующих значений: 10, 20, 30. Если, например, myValue будет равно 10, то на экран выведется строка:

myValue is 10

В языке С и С++ вы можете автоматически перейти к секции следующего case, если в конце предыдущего не стоит инструкция перехода break или goto. Таким образом, на С++ вы можете написать:

```
case 1: statment1;
case 2: statment2;
break;
```

В этом примере на С++ после выполнения statement1 будет автоматически выполняться секция statment2.

В С# это работать не будет. Автоматический переход от case 1 к следующей секции case 2 будет выполняться только в том случае, если секция case 1 окажется пустой (не будет содержать ни одной инструкции). В противном же случае перехода к выполнению case 2 не произойдет, так как в С# каждая непустая секция инструкций оператора case должна содержать в себе оператор break.

```
case 1: Console.WriteLine("Выражение секции 1»);
case 2:
```

Здесь case 1 содержит инструкцию, поэтому вы не сможете автоматически перейти к выполнению case 2. Такой код вообще не будет компилироваться. Если вы хотите перейти после выполнения case 1 к выполнению case 2, то должны явно указать это при помощи оператора goto:

```
case 1: Console.WriteLine("Выражение секции 1");
goto case 2
case 2:
```

Однако другая форма использования оператора switch позволит обойтись без инструкции goto:

```
case 1:
case 2:
Console.WriteLine("Выражение секций 1 и 2»);
```

Такой принцип работы оператора switch используется в случае, когда необходимо выполнить одно и то же действие (или часть действия) для разных значений условного оператора.

```
using System;

namespace SwitchStatement
{
  class MyClass
  {
    static void Main(string[] args)
    {
      int user = 0;
      user = Convert.ToInt32(Console.ReadLine());

      switch(user)
      {
        case 0:
          Console.WriteLine("Здравствуйте User1");
          break;
        case 1:
          Console.WriteLine("Здравствуйте User2");
          break;
        case 2:
          Console.WriteLine("Здравствуйте User3");
          break;
        default:
          Console.WriteLine("Здравствуйте новый пользователь");
          break;
      }
    }
  }
}
```

Этот пример выводит на экран сообщение с приветствием пользователя в зависимости от введенного на экране значения. Так, если вы введете число 1, то на экране появится сообщение «Здравствуйте User2».

В данном примере для выбора выражения, выводящего сообщение на экран, использовалось числовое значение {0, 1 или 2}. Если же вы введете иное значение, нежели представленное в массиве, то на экране появится сообщение «Здравствуйте новый пользователь».

Объявление переменных внутри case инструкций

Рассмотрим случай, когда вам необходимо создать в программе сложную case инструкцию. Для придания программе большей читабельности создание переменных, необходимых для использования, лучше всего объявлять непосредственно перед их применением. Так, если каждый блок case использует свой набор переменных, то и объявление переменных следует делать внутри блоков case. Посмотрите на следующий код:

```
using System;

namespace C_Sharp_Programming
{
  class Part
  {
    public static void Main()
    {
      Console.WriteLine("1: ввод наименования товара\n2: ввод количества
      товара");
      int Choice = Convert.ToInt32(Console.ReadLine());

      switch(Choice)
      {
        case 1:
          string Kane;
          Console.Write("Введите наименование товара " );
          Name = Console.ReadLine();
          break;
        case 2:
          int Count;
          Console.Write("Введите количество товара " );
          Name = Console.ReadLine();
          Count = Convert.ToInt32(Console.ReadLine());
          break;
        default:
          break;
      }
    }
  }
}
```

Знатоки C++ могут без труда увидеть, что на C++ такая switch инструкция компилироваться не будет. Причина этому — объявление переменных внутри блоков case. Однако разработчики C# постарались и предусмотрели возможность создания переменных внутри case блоков. Поэтому на C# такой код является рабочим.

Switch и работа со строками

В приведенном примере переменная user была целочисленного типа. Если вам необходимо использовать в качестве условия оператора switch переменную строкового типа, то вы можете сделать это следующим образом:

```
case "Сергей":
```

Если строк для сравнения много, то по аналогии с целочисленной переменной user используйте несколько инструкций case.

Вот пример использования строковой переменной в качестве условия оператора switch.

```
using System;

namespace SwitchStatement
{
  class MyClass
  {
    static void Main(string[] args)
    {
      string user;
      user = Console.ReadLine();

      switch(user)
      {
        case "user1":
          Console.WriteLine("Здравствуйте пользователь один");
          break;
        case "user2":
          Console.WriteLine ("Здравствуйте пользователь два");
          break;
        case "user3":
          Console.WriteLine ("Здравствуйте пользователь три");
          break;
        default:
          Console.WriteLine("Здравствуйте новый пользователь");
          break;
      }
    }
  }
}
```

В данном случае для идентификации пользователя вам необходимо применить не числовое значение, а строку. Если вы введете строку «user1», то на экране появится сообщение «Здравствуйте пользователь один».

8. ЦИКЛИЧЕСКИЕ ОПЕРАТОРЫ

C# включает достаточно большой набор циклических операторов, таких как for, while, do...while, а также цикл с перебором каждого элемента foreach (плохо знакомый программистам C, но хорошо известный VB-программистам). Кроме того, C# поддерживает операторы перехода и возврата, например goto, break, continue и return.

ОПЕРАТОР goto

Оператор goto был основой для реализации других операторов цикла. Но он был и базой многократных переходов, вследствие чего возникла запутанность кода программы. Поэтому опытные программисты стараются его не использовать, но для того чтобы узнать все возможности языка, рассмотрим и этот оператор.

Он используется следующим образом:

1. Создается метка в коде программы Label1.
2. Организуется переход на эту метку goto Labell.

Имя метки Labell обязательно должно заканчиваться двоеточием. Оно указывает на точку в программе, с которой будет выполняться программа после использования инструкции goto. Обычно инструкция goto привязывается к условию, как показано в следующем примере:

```
using System;
public class Labels
{
 public static int Main( )
  {
 int i = C;
 label:
 Console.WriteLine("i:{0}", i);
 i++;
 if (i < 10)
 goto label;

 return 0;
 }
}
```

Здесь мы выводим на экран строку со значением i десять раз (от 0 до 9). Инструкция goto помогает повторить выполнение одних и тех же инструкций определенное число раз. В этой программе число повторов опре-

деляется инструкцией if(i < 10). Значит, до тех пор пока переменная i будет иметь значение меньшее, чем 10, goto будет переносить нас на метку label:, а значит, вывод строки на экран будет повторяться. То есть с использованием goto мы можем организовать циклический повтор операций в программе.

Именно это явление привело к созданию альтернативного метода организации циклов, такого как while, do..while, for или foreach. Большинство программистов понимают, что использование goto в программе лучше заменять чем-нибудь другим, что приведет к созданию программного кода, более структурированного и понятного, нежели инструкции goto.

ЦИКЛ while

Эта циклическая инструкция работает по принципу: «Пока выполняется условие — происходит работа». Ее синтаксис выглядит следующим образом:

```
while (выражение)
{
  инструкция;
}
```

Как и в других инструкциях, выражение — это условие, которое оценивается как булево значение. Если результатом проверки условия является истина, то выполняется блок инструкций, в противном случае в результате выполнения программы while игнорируется. Рассмотрим пример, приведенный выше, только с использованием while:

```
using System;
public class Labels
{
 public static int Main( )
 {
 int i = 0;
 while(i < 10)
 {
 Console.WriteLine("i: {0}",i) ;
 i++;
 }

 return 0;
 }
}
```

По своей функциональности и та, и другая реализация программы работают абсолютно одинаково, но логика работы несколько изменилась. Заметьте, что цикл while проверяет значение i перед выполнением блока statement. Это гарантирует, что цикл не будет выполняться, если проверяемое условие ложно. Таким образом, если первоначально i примет значение

10 и более, цикл не выполнится ни разу. Инструкция while является вполне самостоятельной, а в данном примере ее можно прочитать подобно предложению: «пока i меньше 10, выводим сообщение на экран и наращиваем i».

ЦИКЛ do... while

Бывают случаи, когда цикл while не совсем удовлетворяет вашим требованиям. Например, вы хотите проверять условие не в начале, а в конце цикла. В таком случае лучше использовать цикл do...while.

```
do{
  инструкция
}while (выражение);
```

Подобно while, выражение — это условие, которое оценивается как булево значение.

Это выражение можно прочитать как: «выполнить действие; если выполняется условие — повторить выполнение еще раз». Заметьте разницу между этой формулировкой и формулировкой работы цикла while. Разница состоит в том, что цикл do...while выполняется всегда минимум один раз, до того как произойдет проверка условия выражения.

```
using System;
public class Labels
{
 public static int Main( )
 {
 int i = 0;

 do

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

 }while(i<10) ;

 return 0;
}}
```

На этом примере видно, что если первоначально i примет значение 10 и более, цикл выполнится. Затем произойдет проверка условия while (i < 10), результатом которой станет ложь (false), и повтора выполнения цикла не произойдет. То есть он выполнится один раз. Как вы помните, при таких же начальных условиях while не выполнился ни разу.

ЦИКЛ for

Если еще раз внимательно посмотреть на примеры (while, do...while, goto), можно заметить постоянно повторяющиеся операции: первоначальная инициализация переменной i, ее наращивание на 1 внутри цикла,

проверка переменной i на выполнение условия (i < 10). Цикл for позволяет вам объединить все операции в одной инструкции.

```
for ( [инициализация ]; [ выражение]; [ наращивание] )
{инструкция}
```

Продолжим рассмотрение на том же примере, но с использованием цикла for:

```
using System;
public class Labels
{
 public static int Main ( )
 {
 for(int i = 0; i < 10; i++]
 {
 Console.WriteLine("i: {0}",i);
 }
 return 0;
}
```

Результатом выполнения такого цикла будет вывод на экран информации вида:

0123456789

Принцип работы такой инструкции очень прост:

1. Происходит инициализация переменной i.
2. Выполняется проверка соответствия условию. Если условие истинно, то происходит выполнение блока вложенных инструкций; если условие оказалось ложным, то цикл прекращается и выполняется программа за фигурными скобками.
3. Наращивается переменная i.

Наращивание переменной внутри цикла происходит на такое число единиц, на которое вы сами зададите. Операция i++ означает «нарастить переменную на 1». Если вы хотите использовать другой шаг изменения i, то смело можете написать так i += 2. В этом случае переменная i будет изменяться на 2 единицы, и на экране вы увидите:

0 2 4 6 8

ЦИКЛ foreach

Эта инструкция незнакома программистам на языке C, она используется для перебора массивов и объединений (collection) по элементам. Разговор о ней пойдет в главе 12.

break И continue

Бывают ситуации, когда необходимо прекратить выполнение цикла досрочно (до того как перестанет выполняться условие) или при каком-

то условии не выполнять описанные в теле цикла инструкции, не прерывая при этом цикла. Для таких случаев очень удобно использовать инструкции break и continue. Если вы хотите на каком-то шаге цикла его прекратить, не обязательно выполняя до конца описанные в нем действия, то лучше всего использовать break. Следующий пример хорошо иллюстрирует его работу.

```
using System;
class Values
{
  static void Main( )
  {
    //объявляем флаг для обозначения простых чисел
    bool IsPrimeNumber;

    for(int i = 100; i > 1; i --)
    {
      //устанавливаем флаг
      IsPrimeNumber = true;

      for (int j = i-1; j > 1; j--)
      {
        //если существует делитель с нулевым остатком
        if(i%j == 0)
        {
          //сбрасываем флаг IsPrimeNumber = false;
        }
      }

      // если не нашлось ни одного делителя
      // с нулевым остатком - то число простое
      if(IsPrimeNumber == true)
        Console.WriteLine("{0} -простое число", i);
    }
  }
}
```

Программа выполняет поиск всех простых чисел от 2 до 100. В программе используется два цикла for. Первый цикл перебирает все числа от 100 до 2. Заметьте, именно от 100 до 2, а не наоборот. Переменная i инициализируется значением 100 и затем уменьшается на 1 с каждой итерацией. Второй цикл перебирает все числа от i до 2. Таким образом, второй цикл будет повторяться 99 + 98 + 97 + ... + 3+2 раз. То есть первый раз он выполнится 99 раз, второй — 98 и т. д. В теле второго цикла проверяется выполнение условия: делится ли число i на число j без остатка (i%j == o). Если это условие верно, то число i нельзя отнести к разряду простых. Следовательно, флажок, определяющий число как простое, устанавливается в false. По окончании работы вложенного цикла проверя-

ется условие — не установился ли флажок, определяющий число как простое, в false. Если нет, то число является простым, и на экран выводится соответствующее сообщение.

В данной программе происходит выполнение всех описанных действий внутри цикла. А что если программа уже отнесла число к разряду не простых чисел? Зачем в этом случае продолжать проверку на существование нулевого делителя? В этом нет необходимости. Это лишь дополнительная загрузка ресурсов программы. Для того чтобы прервать выполнение вложенного цикла, вы можете воспользоваться инструкцией break. Для этого необходимо изменить код второго цикла так, как показано ниже:

```
for(int j = i-1; j > 1; j--)
\
  //если существует делитель с нулевым остатком
  if (i%j == 0]
  {
    //сбрасываем флаг IsPrimeNumber = false;

    // дальнейшая проверка бессмысленна
    break;
  }
}
```

Как только сбросится флаг IsPrimeNumber, вложенный цикл сразу же прервется и выйдет в основной цикл. Таким образом, количество итераций сократится многократно, что благоприятно скажется на производительности работы программы.

Оператор continue в отличие от break не прерывает хода выполнения цикла. Он лишь приостанавливает текущую итерацию и переходит сразу к проверке условия выполнения цикла.

```
for (int j = 0; j < 100; j++)
{
  if (j%2 == 0)

    continue;

} Console.WriteLine("{0}", j);
```

Такой цикл позволит вывести на экран все нечетные числа. Работает он следующим образом: перебирает все числа от 0 до 100. Если очередное число четное — все дальнейшие операции в цикле прекращаются, наращивается число j, и цикл начинается сначала.

СОЗДАНИЕ ВЕЧНЫХ ЦИКЛОВ

При написании приложений с использованием циклов вам следует остерегаться зацикливания программы. Зацикливание — это ситуация, при

которой условие выполнения цикла всегда истинно и выход из цикла невозможен. Давайте рассмотрим простой пример.

```
using System;

namespace C_Sharp_Programming
{
  class Cycles
  {
    public static void Main()
    {
      int n1, r,2;
      r1 = 0;
      r2 = n1 +1;
      while(n1 <n2)
      {
        Console.WriteLine("n1 = {0}, n2 = {1}", n1, n2);
        n1++;
        n2++;
      }
    }
  }
}
```

Здесь условие (nl < n2) всегда истинно. Поэтому выход из цикла невозможен. Следовательно, программа войдет в режим вечного цикла. Такие ошибки являются критическими, поэтому следует очень внимательно проверять условия выхода из цикла.

Однако иногда бывает полезно задать в цикле заведомо истинное условие. Типичным примером вечного цикла является следующая запись:

```
while(true)
(...)
```

«Но ведь такая инструкция приведет к зависанию программы!»,— скажете вы. Да, это возможно, если не задать в теле цикла инструкцию его прерывания. Рассмотрим пример программы:

```
using System;

namespace C_Sharp_Programming
{
  class Cycles
  {
    public static void Main()
    {
      string Name;
      while (true)
      {
        Console.Write("Введите ваш имя ")_;
        Name = Console.ReadLine();
```

```
        Console.WriteLine("Здравствуйте {0}", Name);
      }
    }
  }
}
```

Такая программа не имеет выхода. Что бы не ввел пользователь, программа выдаст строку приветствия и запросит ввод имени заново. Однако все изменится, если в программу добавить условие, при выполнении которого цикл прерывается.

```
using System;
namespace C_Sharp_Programming
{
  class Cycles
  {
    public static void Main()
    {
      string Name;
      while(true)
      {
        Console.Write("Введите ваш имя ");
        Name = Console.ReadLine();

        if (Name == "")
          break;

        Console.WriteLine("Здравствуйте {0} ", Name);
      }
    }
  }
}
```

На этот раз, как только пользователь нажмет клавишу «Enter» без ввода строки данных, сработает инструкция break, и программа выйдет из цикла.

Создание вечных циклов оправдывает себя, если существует несколько условий прерывания цикла и их сложно объединить в одно выражение, записываемое в блоке условия.

Вечный цикл можно создать не только при помощи оператора while. Любой оператор цикла может быть использован для создания вечных циклов. Вот как выглядит та же программа, но с использованием цикла for:

```
using System;

namespace C_Sharp_Programming
{
  class Cycles
  {
    public static void Main()
```

```
    {
      string Name;
      for (;;)
      {
        Console.Write("Введите ваш имя " ) ;
        Name = Console.ReadLine();

        if (Name == "")
          break;

        Console.WriteLine("Здравствуйте {0}", Name);
      }
    }
  }
}
```

Да-да, цикл for может не содержать ни инструкции инициализации, ни инструкции проверки, ни инструкции итерации. Два оператора (;) внутри цикла for означают вечный цикл.

9. КЛАССЫ

Классы — сердце каждого объектно-ориентированного языка. Как вы помните, класс представляет собой инкапсуляцию данных и методов для их обработки. Это справедливо для любого объектно-ориентированного языка, которые отличаются в этом плане лишь типами данных, хранимых в виде членов, а также возможностями классов. В том, что касается классов и многих функций языка, C# кое-что заимствует из C++ и Java и привносит немного изобретательности, помогающей найти элегантные решения старых проблем.

ОПРЕДЕЛЕНИЕ КЛАССОВ

Синтаксис определения классов на C# прост, особенно если вы программируете на C++ или Java. Поместив перед именем вашего класса ключевое слово class, вставьте члены класса, заключенные в фигурные скобки, например:

```
class MyClass
{
private long rr.yClassId;
}
```

Как видите, этот простейший класс с именем MyClass содержит единственный член — myClassId.

НАЗНАЧЕНИЕ КЛАССОВ

Сначала пару слов о том, для чего нужны классы. Представьте себе, что у вас есть некоторый объект, который характеризуется рядом свойств. Например, работник на некой фирме. У него есть такие свойства, как фамилия, возраст, стаж и т. п. Так вот, в этом случае удобно каждого работника описывать не рядом независимых переменных (строкового типа для фамилии, целого типа для возраста и стажа), а одной переменной типа Worker, внутри которой и содержатся переменные для фамилии, возраста и стажа. Здесь самое важное то, что в переменной типа Worker содержатся другие переменные. Конечно, типа Worker среди встроенных типов данных нет, но это не беда — мы можем ввести его.

Еще одна важная вещь: в классах помимо переменных разных типов содержатся функции (или, что то же самое, методы) для работы с этими переменными. Скажем, в нашем примере с классом Worker логично ввести специальные функции для записи возраста и стажа. Функции будут,

в частности, проверять правильность вводимой информации. Например, ясно, что возраст у работника не может быть отрицательным или большим 150. Так вот, наша функция и будет проверять правильность введенного пользователем возраста.

Давайте рассмотрим первый пример класса. Создайте новое консольное приложение для C# и введите следующий текст:

```
using System;
namespace test
{
  //Начало класса
  class Worker
  {
    public int age=0;
    public string name;
  }
  //Конец класса

  class Test
  {
    [STAThread]
    static void Main(string[] args)
    {
      Worker wrk1 = new Worker();
      wrk1.age=34;
      wrk1.name="Иванов";
      Console.WriteLine(wrk1.name+", "+wrk1.age);
    }
  }
}
```

Запустите программу. Она, как и следовало ожидать, выведет на экран *"Иванов, 34"*.

Давайте кратко обсудим наш код. Во-первых, в строчках

```
...
class Worker
{
public int age=0;
public string name;
}
```

мы определили наш класс Worker. Внутри этого класса существуют две переменные — целая age для возраста и name строкового типа для имени. Обратите внимание, что, в отличие от C/C++, мы можем задавать некоторое начальное значение непосредственно сразу после объявления переменной:

```
...
public int age=0;
...
```

Начальное значение задавать вовсе не обязательно — это видно по переменной name.

Перед переменными мы пишем ключевое слово public. Значение у него такое же, как и в C++ — это означает, что переменная (или функция) будет видна вне класса. Если мы не напишем перед переменной никакого модификатора доступа, или укажем private, то переменная не будет видна снаружи класса, и ее смогут использовать только функции этого же класса (т. е. она будет для «внутреннего использования»).

Далее в строчке

```
...
Worker wrk1 = new Worker();
...
```

мы заводим экземпляр класса в куче (heap) и возвращаем на него ссылку.

Затем в строчках

```
wrk1.age=34;
wrk1.name="Иванов";
Console.WriteLine(wrk1.name+", "+wrk1.age);
```

мы используем наш класс, присваивая некоторые значения для возраста и имени и выводя их потом на экран.

СОСТАВ КЛАССОВ

В главе 5 были описаны типы, определенные в CTS (Common Type System). Эти типы поддерживаются как члены классов C# и бывают следующих видов:

- Поле. Так называется член-переменная, содержащий некоторое значение. В ООП поля иногда называют данными объекта. К полю можно применять несколько модификаторов в зависимости от того, как вы собираетесь его использовать. В число модификаторов входят static, readonly и const. Ниже мы познакомимся с их назначением и способами применения.
- Метод. Это реальный код, воздействующий на данные объекта (или поля). Здесь мы сосредоточимся на определении данных класса. Подробнее о методах смотрите в главе 10.
- Свойства. Их иногда называют «разумными» полями (smart fields), поскольку они на самом деле являются методами, которые клиенты класса воспринимают как поля. Это обеспечивает клиентам большую степень абстрагирования за счет того, что им не нужно знать, обращаются ли они к полю напрямую или через вызов метода-аксессора. Подробнее о свойствах смотрите в главе 11.
- Константы. Как можно предположить, исходя из имени, константа — это поле, значение которого изменить нельзя. В главе 4 уже рассматривались константные типы данных. Ниже мы обсудим константы и сравним их с сущностью под названием «неизменяемые поля».

- Индексаторы. Если свойства — это «разумные» поля, то индексаторы — это «разумные» массивы, так как они позволяют индексировать объекты методами-аксессорами get и set. С помощью индексатора легко проиндексировать объект для установки или получения значений. Подробнее — в главе 13.
- События. Событие вызывает исполнение некоторого фрагмента кода. События — неотъемлемая часть программирования для Microsoft Windows. Например, события возникают при движении мыши, щелчке или изменении размеров окна. Об использовании событий смотрите главу 16.

МОДИФИКАТОРЫ ДОСТУПА

Теперь, зная, что типы могут быть определены как члены класса C#, познакомимся с модификаторами, используемыми для задания степени доступа, или доступности данного члена для кода, лежащего за пределами его собственного класса. Они называются модификаторами доступа (access modifiers)

Модификаторы доступа в C#

Модификатор доступа	Описание
public	Член доступен вне определения класса и иерархии производных классов.
protected	Член невидим за пределами класса, к нему могут обращаться только производные классы.
private	Член недоступен за пределами области видимости определяющего его класса. Поэтому доступа к этим членам нет даже у производных классов.
internal	Член видим только в пределах текущей единицы компиляции. Модификатор доступа internal в плане ограничения доступа является гибридом public и protected, зависимым от местоположения кода.

Если вы не хотите оставить модификатор доступа для данного члена по умолчанию (private), задайте для него явно модификатор доступа. Этим C# отличается от C++, где член, для которого явно не указан модификатор доступа, принимает на себя характеристики видимости, определяемые модификатором доступа, заданным для предыдущего члена. Например, в приведенном ниже коде на C++ видимость членов a, Ь и c определена модификатором public, а члены d и e определены как protected:

```
class AccessCPlusPlus
{
public:
int a;
int b;
```

```
int c;
protected:
int d;
int e;
}
```

А в результате выполнения этого кода на C# член b объявляется как private. Для объявления членов на C# как public необходимо использовать следующую инструкцию:

```
class AccessCSharp
{
public int a;
public int b;
public int c;
protected int d;
protectea int e;
}
```

В результате выполнения следующего кода на C# член b объявляется как private:

```
public DifAccessInCSharp
{
public int a;
int b;
}
```

МЕТОД Main

У каждого приложения на C# должен быть метод Main, определенный в одном из его классов. Кроме того, этот метод должен быть определен как public и static (ниже будет объяснено, что значит static). Для компилятора C# не важно, в каком из классов определен метод Main, а класс, выбранный для этого, не влияет на порядок компиляции. Здесь есть отличие от C++, где зависимости должны тщательно отслеживаться при сборке приложения. Компилятор C# достаточно «умен», чтобы самостоятельно просмотреть ваши файлы исходного кода и отыскать метод Main. Между тем, этот очень важный метод является точкой входа во все приложения на C#.

Вы можете поместить метод Main в любой класс, но для его размещения рекомендуется создавать специальный класс. Это можно сделать, используя простой класс MyClass.

```
class MyClass
{
  private int r. MyClassId;
}
class AppClass
{
  static public void Main()
```

```
  {
    MyClass myObj = new MyClass( );
  }
}
```

Как видите, здесь два класса. Этот общий подход используется при программировании на C# даже простейших приложений. MyClass представляет собой класс предметной области, а AppClass содержит точку входа в приложение (Main). В этом случае метод Main создает экземпляр объекта MyClass, и, будь это настоящее приложение, оно использовало бы члены объекта MyClass.

Аргументы командной строки

Вы можете обращаться к аргументам командной строки приложения, объявив метод Main как принимающий аргументы типа массива строк. Затем аргументы могут обрабатываться так же, как любой массив. Хотя речь о массивах пойдет только в главе 12, ниже приводится простой код, который по очереди выводит все аргументы командной строки на стандартное устройство вывода.

```
using System;
class CommandLineApp
{
  public static void Main(string[] args)
  {
    foreach (string arg in args)
    {
      Console.WriteLine( "Аргумент: {0}", arg);
    }
  }
}
```

А вот пример вызова этого приложения с парой случайно выбранных чисел:

```
e:>CommandLineApp 5 42
Аргумент: 5
Аргумент: 42
```

Аргументы командной строки передаются в виде массива строк. Если это флаги или переключатели, их обработку вы должны запрограммировать сами.

Возвращаемые значения

Чаще всего метод Main определяется так:

```
class SomeClass
{
public static void Main()
```

```
{
    . . .
}
    . . .
}
```

Однако вы можете определить метод Main так, чтобы он возвращал значения типа int. Хотя это не является общепринятым в приложениях с графическим интерфейсом, такой подход может быть полезным в консольных приложениях, предназначенных для пакетного исполнения. Оператор return завершает исполнение метода, а возвращаемое при этом значение применяется вызывающим приложением как код ошибки для вывода определенного пользователем сообщения об успехе или неудаче. Для этого служит следующий прототип:

```
public static int Main()
(
// Вернуть некоторое значение типа int,
// представляющее код завершения.
return 0;
}
```

Несколько методов Main

В C# разработчиками включен механизм, позволяющий определять более одного класса с методом Main. Зачем это нужно? Одна из причин — необходимость поместить в ваши классы тестовый код. Затем, используя переключатель /main;<имя_Класса>, компилятору C# можно задавать класс, метод Main которого должен быть задействован. Вот пример, в котором создано два класса, содержащих методы Main:

```
using System;
class Main1
{
  public static void Main()
  {
    Console.WriteLine("Main1");
  }
}
class Main2
{
  public static void Main()
  {
    Console.WriteLine("Main2");
  }
}
```

Чтобы скомпилировать это приложение так, чтобы в качестве точки входа в приложение применялся метод Main1.Main, нужно использовать переключатель:

```
csc MultipleMain.es /raain:Main1
```

При изменении переключателя на /main:Main2 будет использован метод Main2.Main.

Следует соблюдать осторожность и задавать в указанном переключателе имени класса верный регистр символов, так как C# чувствителен регистру. Кроме того, попытка компиляции приложения, состоящего из нескольких классов с определенными методами Main, без указания переключателя /main, вызывает ошибку компилятора.

ИНИЦИАЛИЗАЦИЯ КЛАССОВ И КОНСТРУКТОРЫ

Одно из величайших преимуществ языков ООП, таких как C#, состоит в том, что вы можете определять специальные методы, вызываемые всякий раз при создании экземпляра класса. Эти методы называются конструкторами (constructors). C# вводит в употребление новый тип конструкторов — статические (static constructors), с которыми вы познакомитесь ниже в подразделе «Константы и неизменные поля».

Гарантия инициализации объекта должным образом, прежде чем он будет использован,— ключевая выгода от конструктора. Когда пользователь создает экземпляр объекта, вызывается его конструктор, который должен вернуть управление до того, как пользователь сможет выполнить над объектом другое действие. Именно это помогает обеспечивать целостность объекта и делать написание приложений на объектно-ориентированных языках гораздо надежнее.

Но как назвать конструктор, чтобы компилятор знал, что его надо вызывать при создании экземпляра объекта? Разработчики C# последовали в этом вопросе за разработчиками C++ и провозгласили, что у конструкторов в C# должно быть то же имя, что и у самого класса. Вот простой класс с таким же простым конструктором:

```
using System;
class ConstructorlApp
{
  ConstructorlApp()
  {
    Console. WriteLine("я конструктор.");
  1
  public static void Main()
  (
    ConstructorlAp app = new ConstruotorlApp();
  \
}
```

Значений конструкторы не возвращают. При попытке использовать с конструктором в качестве префикса имя типа компилятор сообщит об ошибке, пояснив, что вы не можете определять члены с теми же именами, что у включающего их типа.

Следует обратить внимание и на способ создания экземпляров объектов в C#. Это делается при помощи ключевого слова new:

```
<класс> <объект> = new <класс>(аргументы конструктора)
```

Если раньше вы программировали на C++, обратите на это особое внимание. В C++ вы могли создавать экземпляр объекта двумя способами: объявлять его в стеке, скажем, так:

```
//Код на C++ Создает экземпляр CMyCleas в стеке
CMyClass rr.yClass;
```

или создавать копию объекта в свободной памяти (или в куче), используя ключевое слово C++ new:

```
//Код на C+-. Создает экземпляр CmyClass в куче.
CMyClass myClass = new CmyClass();
```

Экземпляры объектов на C# формируются иначе, что и сбивает с толку новичков в разработке на C#. Причина путаницы в том, что для создания объектов оба языка используют одни и те же ключевые слова. Хотя с помощью ключевого слова new в C++ можно указать, где именно будет создаваться объект, место его построения на C# зависит от типа объекта, экземпляр которого создается. Как вы уже знаете, ссылочные типы создаются в куче, а размерные — в стеке. Поэтому ключевое слово new позволяет делать новые экземпляры класса, но не определяет место создания объекта.

Хотя можно сказать, что приведенный ниже код на C# не содержит ошибок, он делает совсем не то, что может подумать разработчик на C++:

```
MyClass myClass;
```

На C++ он создаст в стеке экземпляр MyClass. Как сказано выше, на C# вы можете создавать объекты, только используя ключевое слово new. Поэтому на C# эта строка лишь объявляет переменную myClass как переменную типа MyClass, но не создает экземпляр объекта.

Примером служит следующая программа, при компиляции которой компилятор C# предупредит, что объявленная в приложении переменная ни разу не используется:

```
using System;

class Constructor2App
{
  Constructor2App()
  {
    Console.WriteLine("я конструктор");
  }
  public static void Main()
  {
    Constructor2App app;
  }
}
```

Поэтому, объявляя объект, создайте где-нибудь в программе его экземпляр с помощью ключевого слова new:

```
Constructor2App app;
app = new Constructor2App() ;
```

Зачем объявлять объект, не создавая его экземпляров? Объекты объявляются перед использованием или созданием их экземпляров с помощью new, если вы объявляете один класс внутри другого. Такая вложенность классов называется включение (containment) или агрегирование (aggregation).

СТАТИЧЕСКИЕ ЧЛЕНЫ КЛАССА

Вы можете определить член класса как статический (static member) или член экземпляра (instance member). По умолчанию каждый член определен как член экземпляра. Это значит, что для каждого экземпляра класса делается своя копия этого члена. Когда член объявлен как статический, имеется лишь одна его копия. Статический член создается при загрузке содержащего класс приложения и существует в течение жизни приложения. Поэтому вы можете обращаться к члену, даже если экземпляр класса еще не создан. Хотя зачем вам это?

Один из примеров — метод Main. CLR (Common Language Runtime) нужна универсальная точка входа в ваше приложение. Поскольку CLR не должна создавать экземпляры ваших объектов, существуют правила, требующие определить в одном из ваших классов статический метод Main. Вы можете захотеть использовать статические члены при наличии метода, который формально принадлежит классу, но не требует реального объекта. Скажем, если вам нужно отслеживать число экземпляров данного объекта, которое создается во время жизни приложения. Поскольку статические члены «живут» на протяжении существования всех экземпляров объекта, должен работать такой код:

```
using System;

class InstCount
{
  public InstCount()
  {
    instanceCount++;
  }

  static public int instanceCount;
  //instanceCount = 0;
}

class AppClass
{
  public static void Main()
  {
    Console.WriteLine(InstCount.instanceCount);
    InstCount ic1 = new InstCount();
```

```
    Console.WriteLine(InstCount.instanceCount);

    InstCount ic2 = new InstCount ();
    Console.WriteLine(InstCount.instanceCount) ;
  }
}
```

В этом примере выходная информация будет следующая:

```
0
1
2
```

И последнее замечание по статическим членам: у них должно быть некоторое допустимое значение. Его можно задать при определении члена:

```
static public int instanceCountt = 10;
```

Если вы не инициализируете переменную, это сделает CLR после запуска приложения, установив значение по умолчанию, равное 0. Поэтому следующие строки эквивалентны:

```
static public int instanceCount2;
static public int instanceCount2=0;
```

КОНСТАНТЫ И НЕИЗМЕНЯЕМЫЕ ПОЛЯ

В главе 5 уже упоминалось о константах. Здесь лишь еще раз будет описана их специфика и дана сравнительная характеристика с неизменяемыми полями. Можно с уверенностью сказать, что возникнут ситуации, когда изменение некоторых полей при выполнении приложения будет нежелательно, например, это могут быть файлы данных, от которых зависит ваше приложение, значение pi для математического класса или любое другое используемое в приложении значение, о котором вы знаете, что оно никогда не изменится. В этих ситуациях C# позволяет определять члены тесно связанных типов: константы и неизменяемые поля.

Константы

Из названия легко догадаться, что константы (constants), представленные ключевым словом const,— это поля, остающиеся постоянными в течение всего времени жизни приложения. Определяя что-либо как const, достаточно помнить два правила. Во-первых, константа — это член, значение которого устанавливается в период компиляции программистом или компилятором (в последнем случае это значение по умолчанию). Во-вторых, значение члена-константы должно быть записано в виде литерала.

Чтобы определить поле как константу, укажите перед определяемым членом ключевое слово const:

```
using System;

class MagicNumbers
```

```
{
  public const double pi = 3.1415;
  public const int [g] = (10) ;
}

class ConstApp
{
  public static void Main()
  {
    Console.WriteLine("pi = {0}, g = {1}",
      MagicNumbers.pi,MagicNumbers.[g]);
  }
}
```

Обратите внимание на один важный момент, связанный с этим кодом. Клиенту нет нужды создавать экземпляр класса MagicNumbers, поскольку по умолчанию члены const являются статическими.

Неизменяемые поля

Поле, определенное как const, ясно указывает, что программист намерен поместить в него постоянное значение. Это плюс. Но оно работает, только если известно значение подобного поля в период компиляции. А что же делать программисту, когда возникает потребность в поле, чье значение не известно до периода выполнения, но после инициализации не должно меняться? Эта проблема (которая обычно остается нерешенной в большинстве других языков) разрешена разработчиками языка C# с помощью неизменяемого поля (read-only field).

Определяя поле с помощью ключевого слова readonly, вы можете установить значение поля лишь в одном месте — в конструкторе. После этого поле не могут изменить ни сам класс, ни его клиенты. Допустим, для графического приложения нужно отслеживать разрешение экрана. Справиться с этой проблемой с помощью const нельзя, так как до периода выполнения приложение не может определить разрешение экрана у пользователя. Поэтому лучше всего использовать такой код:

```
using System;

class GraphicsPackage
{
  public readonly int ScreenWidth;
  public readonly int ScreenHeight;

  public GraphicsPackage()
  {
    this.ScreenWidth = 1024;
    this.ScreenHeight = 768;
  }
}
```

```
class ReadOnlyApp
{
  public static void Main()
  {
    GraphicsPackage graphics = new GraphicsPackage();
    Console.WriteLine("Ширина = {0}, Высота = {1}",
      graphics.ScreenWidth,
      graphics.ScreenHeight);
  }
}
```

На первый взгляд кажется: это то, что нужно. Но есть одна маленькая проблема: определенные нами неизменяемые поля являются полями экземпляра, а значит, чтобы задействовать их, пользователю придется создавать экземпляры класса. Может, это и не проблема, и код даже пригодится, когда значение неизменяемого поля определяется способом создания экземпляра класса. Однако если вам нужна константа, по определению статическая, но инициализируемая в период выполнения? Тогда нужно определить поле с обоими модификаторами — static и readonly, а затем создать особый, статический тип конструктора. Статические конструкторы (static constructor) используются для инициализации статических, неизменяемых и других полей. Здесь был изменен предыдущий пример так, чтобы сделать поля, определяющие разрешение экрана, статическими и неизменяемыми, а также добавлен статический конструктор. Обратите внимание на ключевое слово static, добавленное к определению конструктора:

```
using System;

class ScreenResolution
{
  public static readonly int ScreenWidth;
  public static readonly int ScreenHeight;

  static ScreenResolution()
  {
    // code would be here to
    // calculate resolution
    ScreenWidth = 1024;
    ScreenHeight = 768;
  }
}

class ReadOnlyApp
{
  public static void Main()
  {
```

```
    Console.WriteLine("Ширина = {0}, Высота = {1}",
      ScreenResolution.ScreenWidth,
      ScreenResolution.ScreenHeight);
  }
}
```

ВЛОЖЕННЫЕ КЛАССЫ

Иногда некоторый класс играет чисто вспомогательную роль для другого класса и используется только внутри него. В этом случае логично описать его внутри существующего класса. Вот пример такого описания:

```
using System;
namespace test
{
  class ClassA
  {
    //Вложенный класс
    private class ClassB
    {
      public int z;
    }
    //Переменная типа вложенного класса
    private ClassB w;
    //Конструктор
    public ClassAO
    {
      w=new ClassB();
      w.z=35;
    }
    //Некоторый метод
    public int SomeMethod()
    {
      return w.z;
    }
  }
  class Test
  {
    static void Main(string[] args)
    {
      ClassA v=new ClassAO;
      int k=v.SomeMethod();
      Console.WriteLine(k);
    }
  }
}
```

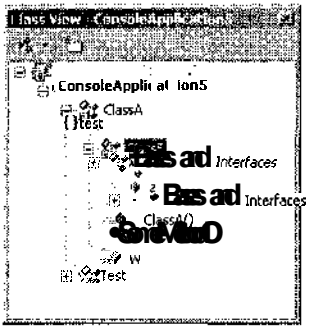


Рис. 9.1. Отображение вложенных классов

После запуска программа выведет результат:
55

Здесь класс ClassB объявлен внутри класса ClassA, причем со словом private, так что его экземпляры мы можем создавать только внутри класса ClassA (что мы и делаем в конструкторе класса ClassA). Методы класса ClassA имеют доступ к экземпляру класса ClassB (как, например, метод SomeMethod).

Вложенный класс имеет смысл использовать тогда, когда его экземпляр используется только в определенном классе. Кроме того, с вложением классов улучшается читаемость кода — если нас не интересует устройство основного класса, то разбирать работу вложенного класса нет необходимости.

Обратите также внимание, как вложенные классы показываются на вкладке ClassView (см. рис. 9.1).

НАСЛЕДОВАНИЕ

Чтобы построить (в терминах данных или поведения) класс на основе другого класса, применяется наследование. Оно принадлежит к правилам заменяемости, а именно к тому, которое гласит, что производный класс может быть заменен базовым. Примером этого может служить написание иерархии классов базы данных. Допустим, вам был нужен класс для обработки баз данных Microsoft SQL Server и Oracle. Поскольку эти базы различны, вам понадобится по классу для каждой из БД. Однако в обеих БД достаточная доля общей функциональности, которую можно поместить в базовый класс, а два других класса сделать его производными, при необходимости подменяя или изменяя поведение, унаследованное от базового класса.

Чтобы унаследовать один класс от другого, используется синтаксис:

```
class <производный_класс>: <базовый_класс>
```

Вот пример того, на что может быть похожа такая программа с базами данных:

```
using System;

class Database
{
  public Database))
  {
    CommonField = 42,-
  }

  public int CommonField;
```

```
  public void CommonMethod()
  {
    Console.WriteLine("Database.Common Method");
  }
}

class SQLServer: Database
{
  public void SomeMethodSpecificToSQLServer()
  {
    Console.WriteLine("SQLServer.SomeMethodSpecificToSQLServer");
  }
}

class Oracle: Database
{
  public void SomeMethodSpecificToOracle()
  {
    Console.WriteLine("Oracle.SomeMethodSpecificToOracle");
  }
}

class InheritanceApp
{
  public static void Main()
  {
    SQLServer sqlserver = new SQLServer();

    sqlserver.SomeMethodSpecificToSQLServer();
    sqlserver.CommonMethod();
    Console.WriteLine("Inherited common field = {0} ", sqlserver.CommonField);
  }
}
```

После компиляции и исполнения этого приложения получится следующий результат:

SQLServer,SomeMethodSpecificToSQLServer Database.Common Method
Inherited common field =42

Заметьте: методы Database.CommonMethod и Database.CommonField теперь являются частью определения класса SQLServer. Так как классы SQLServer и Oracle происходят от базового класса Database, оба наследуют практически все его члены, определенные как public, protected или internal. Единственное исключение — конструктор, который не может быть унаследован. У каждого класса должен быть реализован собственный конструктор, независимый от базового класса.

Подмена методов описана в главе 10. Однако следует заметить, что подмена методов позволяет изменять реализацию унаследованных от

базового класса методов. Абстрактные классы интенсивно используют подмену методов (см. ниже).

Для наглядности давайте рассмотрим еще один пример. Представьте, что у нас есть некоторый класс (быть может, весьма большой), который почти подходит для нашей конкретной задачи. Почти, но не совсем. Что-то (некоторые методы) в этом классе надо изменить, что-то — добавить. Наследование как раз и служит для этого. При наследовании мы объявляем наш класс потомком другого класса. И наш класс-потомок (или, как его еще называют, дочерний класс) автоматически приобретает все, что было в родительском классе. Это мы делаем буквально парой строчек. Затем в дочерний класс мы можем что-нибудь добавить (чего не было в классе родительском). Кроме того, в дочернем классе мы можем что-нибудь из родительского класса переопределить — так, что оно уже будет работать по-другому.

Например, у нас будет класс Worker и производный от него класс Boss. Класс Boss будет отличаться от класса Worker, во-первых, наличием переменной numOfWorkers (для количества подчиненных) и, во-вторых, другой работой в Boss метода для задания возраста (setAge):

```
using System;
namespace test
{
  //Класс Worker
  class Worker
  {
    protected int age=0;
    public void setAge(int age)
    {
      if(age>0 && age<100)
        this.age=age;
      else
        this.age=0;
    }
    public int getAge()
    {
      return age;
    }
  }
  //Класс Boss
  class Boss: Worker
  {
    public int numOfWorkers; //Количество подчиненных
    public new void setAge(int age)
    {
      if(age>0 && age<45)
        this.age=age;
```

```
      else
        this.age=0;
    }
  }
  class Test.
  {
    static void Main(string[] args)
    {
      Worker wrk1 = new Worker ();
      Boss boss = new Boss () ;
      wrk1.setAge(50) ;
      boss.setAge(50);
      boss.numOfWorkers=4;
      Console.WriteLine("Возраст работника "+wrk1.getAge());
      Console.WriteLine("Возраст босса "+boss.getAge()+
        ".\nКоличество подчиненных "+boss.numOfWorkers);
    }
  }
}
```

Обратите внимание, как мы вводим новый класс Boss:

```
...
 class Boss: Worker{
...
```

Такая запись и означает, что новый класс Boss будет потомком другого класса (Worker в данном случае) и, следовательно, будет автоматически уметь все то же самое, что и родительский класс. В частности, в нем есть переменная age для хранения возраста. Далее, в класс Boss мы вводим переменную numOfWorkers для хранения количества подчиненных у нашего босса. В родительском классе такой переменной не было. Кроме того, нас не устроила реализация метода setAge из родительского класса Worker (по каким-то внутренним инструкциям фирмы возраст босса не может превышать 45 лет, тогда как возраст работника не может превышать 100), поэтому мы просто написали в классе Boss метод с таким же именем. Обратите внимание на слово new при объявлении метода:

```
...
 public new void setAge(int age)
...
```

Таким образом, в производном классе мы можем что-нибудь добавлять и что-нибудь изменять по отношению к классу родительскому. При этом убирать что-нибудь полученное в наследство от родительского класса мы не можем.

После запуска наша программа выдаст вполне ожидаемый результат:

Возраст работника 50
Возраст босса О
Количество подчиненных 4

ИНИЦИАЛИЗАТОРЫ КОНСТРУКТОРОВ

Во всех конструкторах C#, кроме System.Object, конструкторы базового класса вызываются прямо перед исполнением первой строки конструктора. Эти инициализаторы конструкторов позволяют задавать класс и подлежащий вызову конструктор. Они бывают двух видов.

Инициализатор в виде base(...) активизирует конструктор базового класса текущего класса.

Инициализатор в виде this(...) позволяет текущему базовому классу вызвать другой конструктор, определенный в нем самом. Это полезно, когда вы перегрузили несколько конструкторов и хотите быть уверенными, что всегда будет вызван конструктор по умолчанию. О перегруженных методах смотрите главу 10, здесь же мы приведем их краткое определение: перегруженными называются два и более метод с одинаковым именем, но с различными списками аргументов.

Чтобы увидеть порядок событий в действии, обратите внимание на следующий код: он сначала исполнит конструктор класса A, а затем конструктор класса B:

```
using System;

class Л
{
  public A()
  !
     Console.WriteLine("A");
   }
}

class B: A
{
public B() ; base ()
  {
     Console.WriteLine("B");
   1
'

class BaseDefaultInitializerApp
{
  public static void Main()
  {
    B b - new B() ;
  }
}
```

Этот код — функциональный эквивалент следующего, где производится явный вызов конструктора базового класса:

```
using System;

class A
{
  public A()
  {
    Console.WriteLine("A");
  }
)

class B: A
{
  public B()
  {
    Console.WriteLine("B");
  }
}

class DefaultInitializerApp
[
  public static void Main()
  {
    B b •- new B() ;
  )
}
```

А теперь рассмотрим более удачный пример ситуации, когда выгодно использовать инициализаторы конструкторов. У нас опять два класса: A и B. На этот раз у класса A два конструктора, один из них не требует аргументов, а другой принимает аргумент типа int. У класса B один конструктор, принимающий аргумент типа int. При создании класса B возникает проблема. Если запустить следующий код, будет вызван конструктор класса A, не принимающий аргументов:

```
using System;

class A
{
  public A()
  {
    Console.WriteLine("A");
  }

  public A(int foo)
  {
    Console.WriteLine("A = {0}", foo);
  }
}
```

```
class B: A
{
  public B(int too]
  {
    Console.WriteLine("B = {0)", foo);
  }
}

class DerivedInitializer1App
i
  public static void Main])
  {
    B b = new B(42);
  }
}
```

Как же гарантировать, что будет вызван именно нужный конструктор класса A? Явно указав компилятору, какой конструктор в инициализаторе должен быть вызван первым, скажем, так:

```
using System;

class A
{
  public A()
  {
    Console.WriteLine("A");
  }
  public A(int foo)
  {
    Console.WriteLine("A = {0}", foo);
  }
}

class B: A
{
  public B(int foo): base (foo)
  {
    Console.WriteLine("B = {0}", foo];
  }
}

class DerivedInitializer2App
(
  public static void Main()
  {
```

```
    B b = new B(42);
  }
}
```

Использование интерфейсов

C# не поддерживает множественное наследование. Но вы можете объединять характеристики поведения нескольких программных сущностей, реализовав несколько интерфейсов. Об интерфейсах и способах работы с ними смотрите главу 15. Пока можете думать об интерфейсах как о разновидности классов.

С учетом этого следующая программа ошибочна:

```
class Foo
{
}
class Bar
{
}
class MyClass: Foo, Bar
{
  public static void Main ()
    {
    }
}
```

Допущенная в этом примере ошибка связана со способом реализации интерфейсов. Интерфейсы, выбранные для реализации, перечисляются после базового класса данного класса. В этом примере компилятор C# думает, что Bar должен быть интерфейсом. Поэтому компилятор C# выдаст вам сообщение об ошибке:

'Bar' -type in interface list is not an interface

Следующий, более реалистичный, пример абсолютно верен, так как класс MyClass происходит от Control и реализует интерфейсы IFoo к IBar:

```
class Control
{
}
interface IFoo
{
}
interface IBar
{
}
class MyClass: Control, IFoo, IBar
{
}
```

Главное здесь то, что единственным способом реализации чего-либо, напоминающего множественное наследование, на C# является применение интерфейсов.

Изолированные классы

Если вы хотите быть уверены, что класс никогда не будет использован как базовый, при определении класса примените модификатор sealed. Единственное ограничение: абстрактный класс не может быть изолированным, так как в силу своей природы предназначен для использования в качестве базового. И еще. Хотя изолированные классы предназначены для предотвращения непреднамеренного создания производных классов, определение класса как изолированного позволяет выполнить оптимизацию периода выполнения. В частности, поскольку компилятор гарантирует отсутствие у класса любых производных классов, есть возможность преобразования виртуальных вызовов для изолированного класса в невиртуальные вызовы. Вот пример определения класса как изолированного:

```
using System;

sealed class MyPoint
{
  public MyPoint(int x, int y)
  {
    this.x = x;
  } this.y = y;

  private int X;
  public int x
  {
    get
    {
      return this.X;
    }
    set
    {
      this.X = value;
    }
  }

  private int Y;
  public int y
  {
    get
    {
      return this.Y;
    }
    set
    {
```

```
      this.Y = value;
    }
  }
}

class SealedApp
{
  public static void Main()
  {
    MyPoint pt = new MyPoint(6,16);
    Console.WriteLine("x = {0}, y = {1}", pt.x, pt.y);
  }
}
```

Здесь использован модификатор доступа private для внутренних членов класса X и Y. В результате применения модификатора protected компилятор выдаст предупреждение, так как защищенные члены видимы производным классам, а, как вам известно, у изолированных классов производных классов нет.

АБСТРАКТНЫЕ КЛАССЫ

Методы класса могут быть объявлены как абстрактные. Это означает, что в этом классе нет реализации этих методов. Абстрактные методы пишутся с модификатором abstract. Класс, в котором есть хотя бы один абстрактный метод, называется абстрактным (в таком классе могу быть и обычные методы). Нельзя создавать экземпляры абстрактного класса — такой класс может использоваться только в качестве базового класса для других классов. Для потомка такого класса есть две возможности — или он реализует все абстрактные методы базового класса (и в этом случае для такого класса-потомка мы сможем создавать его экземпляры), или реализует не все абстрактные методы базового класса (в этом случае он является тоже абстрактным классом, и единственная возможность его использования — это производить от него классы-потомки). Вот пример, иллюстрирующий использование абстрактных классов:

```
using System;
namespace test
{
  abstract class Figure
  {
    //Площадь фигуры
    public abstract double square();
    public abstract double perimeter();
  }
  class Triangle: Figure
```

```
{
  double a, b, c; //Стороны
  //Конструктор
  public Triangle(double a, double b, double c)
  {
    this.a=a;
    this.b=b;
    this.c=c;
  }
  public override double square()
  {
    //Используем формулу Герона
    double p = (a+b+c)/2;
    return Math.Sqrt(p*(p-a)*(p-b)*(p-c));
  }
  public override double perimeter()
  {
    return a+b+c;
  }
}
class Rectangle: Figure
{
  double a, b; //Стороны
  //Конструктор
  public Rectangle(double a, double b)
  {
    this.a=a;
    this.b=b;
  }
  public override double square()
  {
    return a*b;
  }
  public override double perimeter()
  {
    return (a+b)*2;
  }
}
class Test
{
  public static void Main()
  {
    Figure f1, f2;
    f1=new Triangle(3, 4, 5);
    f2=new Rectangle(2, 6);
    System.Console.WriteLine(f1.perimeter()+", "+ f1.square());
```

```
        System.Console.WriteLine(f2.perimeter()+", "+f2.square())
    }
  }
}
```

Результаты работы программы отобразятся как:

12, 6

16, 12

Тут мы объявляем абстрактный класс Figure, от которого производим два класса — Rectangle (класс прямоугольника) и Triangle (треугольника). В классе Figure есть два абстрактных метода — square (для подсчета площади) и perimeter (для периметра). Так как для произвольной фигуры формул для площади и для периметра не существует, то эти методы объявлены в классе Figure и переопределены в производных классах (с ключевым словом override). Далее в классе Test мы проводим испытание — заводим две переменные ссылочного типа базового класса Figure, ниже в эти ссылки мы записываем созданные экземпляры производных классов Triangle и Rectangle. Обратите внимание, что ссылку на абстрактный класс мы создать можем, а экземпляр — нет. Далее мы выводим на экран периметр и площадь для наших фигур.

10. МЕТОДЫ

Как вы уже знаете, классы — это инкапсулированные наборы данных и методов, обрабатывающих их. Иначе говоря, методы определяют поведение классов. Мы называем методы в соответствии с действиями, выполняемыми классами по нашему желанию в наших интересах. До сих пор мы не вдавались в подробности определения и вызова методов на C#. Вот этому и будет посвящена данная глава: вы узнаете о ключевых словах параметров методов ref и out, а также о том, как с их помощью определяются методы, которые возвращают вызывающему коду больше одного значения. Кроме того, вы научитесь определять перегруженные методы — когда несколько методов с одинаковым именем могут по-разному функционировать в зависимости от типов и числа переданных им аргументов. Затем вы узнаете, как поступать в ситуациях, когда до момента выполнения неизвестно точное число аргументов метода. В завершение мы рассмотрим виртуальные методы, основы наследования и способы определения статических методов.

Метод — это именованная часть программы, к которой можно обращаться из других частей программы столько раз, сколько потребуется. Рассмотрим программу, выводящую на экран квадраты чисел:

```
using System;
class Degrees
{
 public static int Square(int x)
 {
 return x*x;
 }
 public static void Main()
 {
 for(int i = 0; i < 10;                    i++)

 Console.WriteLine(Square(i));
 }
}
```

Здесь объявлена функция Square. При вызове функции программа прерывает ход основной программы и переходит к выполнению кода, описанного в функции. После окончания работы функции управление передается вызывающей функции.

Функция Square принимает значения типа int. При вызове тип параметра функции сопоставляется с ожидаемым типом точно так же, как если

бы инициализировалась переменная описанного типа. Это гарантирует надлежащую проверку и преобразование типов. Например, обращение Square("abcd") вызовет «недовольство» компилятора, поскольку «abcd» является строкой, а не int.

Определение функции задает имя функции, тип возвращаемого ею значения (если таковое имеется), типы и имена ее параметров (если они есть). Значение возвращается из функции с помощью оператора return. Разные функции обычно имеют разные имена, но функциям, выполняющим сходные действия над объектами различных типов, иногда лучше дать возможность иметь одинаковые имена. Если типы их параметров различны, то компилятор всегда сможет различить их и выбрать для вызова нужную функцию. Может, например, иметься одна функция возведения в степень для целых переменных и другая для переменных с плавающей точкой.

Если функция не возвращает значения, то ее следует описать как void. Например, как функция Main в нашем примере.

ПЕРЕДАЧА ПАРАМЕТРОВ

При попытке добыть информацию с помощью метода на C# вы получите только возвращаемое значение. Поэтому может показаться, что в результате вызова метода вы получите не более одного значения. Очевидно, что отдельный вызов метода для каждой порции данных во многих ситуациях будет очень неуклюжим решением. Допустим, у вас есть класс Color, представляющий любой цвет в виде трех значений согласно стандарту RGB (красный-зеленый-синий). Использование лишь возвращаемых значений вынудит вас написать следующий код, чтобы получить все три значения:

```
// Предполагаем, что color - экземпляр класса Color,
int red = color.GetRed();
int green = color.GetGreen();
Int blue = color.GetBlue();
```

Но нам хочется получить что-то вроде этого:

```
int red;
Int green;
int blue;
color. GetRGB (red, green, blue);
```

Но при этом возникает проблема. При вызове метода color.GetRGB значения аргументов red, green и blue копируются в локальный стек метода, а переменные вызывающей функции остаются без изменений, сделанных методом.

На C++ эта проблема решается путем передачи при вызове метода указателей или ссылок на эти переменные, что позволяет методу обрабатывать данные вызывающей функции. Решение на C# выглядит аналогично. На самом деле C# предлагает два похожих решения. Первое из

них использует ключевое слово ref. Оно сообщает компилятору C#, что передаваемые аргументы указывают на ту же область памяти, что и переменные вызывающего кода. Таким образом, если вызванный метод изменяет их и возвращает управление, переменные вызывающего кода также подвергнутся изменениям. Следующий код иллюстрирует использование ключевого слова ref на примере класса Color.

```
using System;

class Color
{
  public Color ()
  {
    this.red = 255;
    this.green = 0;
    this.blue = 125;
  }

  protected int red;
  protected int green;
  protected int blue;

  public void GetColors (ref int red, ref int green, ref int blue)
  {
    red = this.red;
    green = this.green;
    blue = this.blue;
  }
}

class RefTest1App
{
  public static void Main()
  {
    Color color = new Color ();
    int red;
    int green;
    int blue;
    color.GetColors(ref red, ref green, ref blue);
    Console.WriteLine("красный = {0}, зеленый = {1}, синий = {2}",
      red, green, blue);
  }
}
```

Приведенный код не будет компилироваться. У ключевого слова ref есть одна особенность: перед вызовом метода вы должны инициализировать передаваемые аргументы. Поэтому, чтобы этот код заработал, его нужно изменить:

```
using System;

class Color
{
  public Color()
  {
    this.red = 255;
    this.green = 0;
    this.blue = 125;
  }

  protected int red;
  protected int green;
  protected int blue;

  public void GetColors(ref int red, ref int green, ref int blue)
  {
    red = this.red;
    green = this.green;
    blue = this.blue;
  }
}

class RefTest1App
{
  public static void Main()
  {
    Color color = new Color();
    int red = 0;
    int green = 0;
    int blue = 0;
    color.GetColors (ref red, ref green, ref blue);
    Console.WriteLine("красный - {0}, зеленый = {1}, синий = {2}",
      red, green, blue);
  i
}
```

Не кажется ли вам, что инициализация переменных, которые позже будут перезаписаны, бессмысленна? Поэтому C# предоставляет альтернативный способ передачи аргументов, изменения значений которых должны быть видимы вызывающему коду: с помощью ключевого слова out. Вот пример с тем же классом Color, где используется out:

```
using System;

class Color
{
  public Color 0
```

```
  {
    this.red = 255;
    this.green = 0;
    this.blue = 125;
  }

  protected int red;
  protected int green;
  protected int blue;

  public void GetColors(out int red, out int green, out int blue)
  {
    red = this.red;
    green = this.green;
    blue = this.blue;
  }
}

class OutTest1App
{
  public static void Main()
  {
    Color color = new Color();
    int red;
    int green;
    int blue;

    color.GetColors(out red, out green, out blue);
    Console.WriteLine("red = {0}, green = {1}, blue = {2}",
      red, green, blue);
  }
}
```

Единственное различие между ключевыми словами ref и out в том, что out не требует, чтобы вызывающий код сначала инициализировал передаваемые аргументы. А когда же применять ref? Когда нужна гарантия, что вызывающий код инициализировал аргумент. В приведенных примерах можно было применять ключевое слово out, так как вызываемый метод не зависит от значения передаваемой переменной. Ну а если вызываемый метод использует значение параметра? Взгляните:

```
using System;

class Window
{
  public Window (int x, int y)
```

```
  {
    this.x = x;
    this.y = y;
  }

  protected int x;
  protected int y;

  public void Move(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public void ChangePos(ref int x, ref int y)
  {
    this.x += x;;
    this.y += y;

    x = this.x;
    y = this.y;
  }
}

class OutTest2App
{
  public static void Main()
  {
    Window wnd = new Window(5, 5);
    int x = 5;
    int y = 5;

    wnd.ChangePos(ref x, ref y);
    Console.WriteLine("{0}, {1}", x, y);

    x = -1;
    y = -1;
    wnd.ChangePos(ref x, ref y);
    Console.WriteLine("{0}, {1}", x, y);
  }
}
```

Как видите, работа вызываемого метода Window.ChangePos основана на переданных ему значениях. В данном случае ключевое слово ref вынуждает вызывающий код инициализировать эти значения, чтобы метод работал корректно.

ПЕРЕГРУЗКА МЕТОДОВ

Перегрузка методов позволяет программистам на C# многократно использовать одни и те же имена методов, меняя лишь передаваемые аргументы. Это очень полезно, по крайней мере, в двух сценариях. Первый. Вам нужно иметь единое имя метода, поведение которого немного различается в зависимости от типа переданных аргументов. Допустим, у вас есть класс, отвечающий за протоколирование и позволяющий вашему приложению записывать на диск диагностическую информацию. Чтобы немного повысить гибкость класса, вы можете создать несколько форм метода Write, определяющих тип записываемой информации. Кроме собственно строки, подлежащей записи, метод также может принимать строку идентификатора ресурса. Без перегрузки методов вам пришлось бы реализовать отдельные методы наподобие WriteString и WriteFromResourceid для каждой ситуации. Однако перегрузка методов позволяет реализовать оба метода под именем WriteEntry, при этом они будут различаться лишь типом параметра:

```
using System;

class Log
{
  public Log(string fileName)
  {
    // Open fileName and seek to end.
  }

  public void WriteEntry(string entry)
  {
    Console.WriteLine(entry);
  }

  public void WriteEntry(int resourceId)
  {
    Console.WriteLine
      ("получить строку по id ресурса и вывести в log");
  }
}

class Overloading1App
{
  public static void Main()
  {
    Log log = new Log("My File");
    log.WriteEntry("Строка один");
    log.WriteEntry(42);
  }
}
```

Во втором сценарии выгодно применять перегрузку метода — использование конструкторов, которые, в сущности, представляют собой методы, вызываемые при создании экземпляра объекта. Допустим, вы хотите создать класс, который может быть построен несколькими способами. Например, он использует описатель (int) или имя (string) файла, чтобы открыть его. Поскольку правила C# диктуют, что у конструктора класса должно быть такое же имя, как и у самого класса, вы не можете просто создать разные методы для переменных каждого типа. Вместо этого нужно использовать перегрузку конструктора:

```
using System;

class File
{
}

class CommaDelimitedFile
{
  public CommaDelimitedFile(String fileName)
  {
    Console.WriteLine("Constructed with a file name");
  }

  public CommaDelimitedFile(File file)
  {
    Console.WriteLine("Constructed with a file object");
  }
}

class Overloading2App
{
  public static void Main()
  {
    File file = new File();
    CommaDelimitedFile file2 = new CommaDelimitedFile(file);
    CommaDelimitedFile file3 =
      new CommaDelimitedFile("Some file name");
  }
}
```

О перегрузке метода важно помнить следующее: список аргументов каждого метода должен отличаться. Поэтому следующий код не будет компилироваться, так как единственное различие между двумя версиями Overloading3App.Foo — тип возвращаемого значения:

```
using System;

class Overloading3App
{
  void Foo(double input)
```

```
  {
    Console.WriteLine("Overloading3App.Foo(double)");
  }

  // ОШИБКА: методы отличаются лишь типом возвращаемого значения.
  // Код компилироваться не будет
  double Foo(double input)
  {
    Console.WriteLine("Overloading3App.Foo(double)  (second version)");
  }

  public static void Main()
  (
    Overloading3App app = new Overloading3App();

    double i = 5;
    app.Foo(i);
  }
}
```

ПЕРЕМЕННОЕ ЧИСЛО ПАРАМЕТРОВ

Иногда число аргументов, которые будут переданы методу, неизвестно до периода выполнения. Например, вам нужен класс, который чертит на графике линию, заданную последовательностями x- и y- координат. У вас может быть класс с методом, принимающим единственный аргумент,— объект Point, который представляет значения обеих координат x и y. Затем этот метод сохраняет каждый объект Point в связанном списке или в элементах массива, пока вызывающая программа не даст команду на вывод всей последовательности точек. Однако это решение не годится по двум причинам. Во-первых, оно вынуждает пользователя делать рутинную работу по вызову одного и того же метода для каждой точки линии, которую нужно нарисовать (очень утомительно, если линия будет длинной), а чтобы нарисовать линию, затем приходится вызывать другой метод. Во-вторых, данное решение требует от класса хранить эти точки, хотя они нужны лишь для того, чтобы использовать единственный метод — DrawLine. Эту проблему эффективно решает переменное количество аргументов.

Вы можете задавать переменное число параметров метода через ключевое слово params и указание массива в списке аргументов метода. Вот пример класса Draw, написанного на C#, который позволяет пользователю одним вызовом получить и вывести произвольное число объектов Point:

```
using System;

class Point
{
  public Point(int x, int y)
```

```
  {
    this.x = x;
    this.y = y;
  }

  public int x;
  public int y;
}

class Chart
{
  public void DrawLine(params Point[] p)
  {
    Console.WriteLine("\n Этот метод позволяет нарисовать линию " +
      "по следующим точкам: " ) ;
    for (int i = 0; i < p.GetLength(0); i++)
    {
      Console.WriteLine("{0}, (1) ", p[i].x, p[i] .y);
    }
  }
}

class VarArgsApp
{
  public static void Main()
  {
    Point p1 = new Point(5,10);
    Point p2 = new Point(5, 15);
    Point p3 = new Point(5, 20);

    Chart chart = new Chart();
    chart.DrawLine(p1, p2, p3);
  }
}
```

Метод DrawLine сообщает компилятору C#, что он может принимать переменное число объектов Point. Затем в период выполнения метод использует простой цикл for для прохода по всем объектам Point и вывода всех точек.

В реальном приложении для доступа к членам x и y объекта Point намного лучше будет использовать свойства, чем объявлять эти члены как public. Кроме того, в методе DrawLine было бы лучше применять оператор foreach вместо цикла for. (О свойствах будет рассказано в главе 11, а об операторе foreach — в главе 12.)

Вы уже узнали из главы 9, что можете производить один класс от другого, при этом новый класс будет наследовать возможности уже существующего класса. Еще ничего не зная о методах, мы лишь вскользь

коснулись наследования полей и методов. Иначе говоря, мы еще не рассматривали возможности изменения поведения производных классов. А делается это с помощью виртуальных методов.

ПОДМЕНА МЕТОДОВ

Давайте сначала рассмотрим способы подмены (override) функциональности базового класса в унаследованном методе. Начнем с базового класса, представляющего сотрудника. Чтобы максимально упростить пример, у этого класса будет единственный метод — CalculatePay, который будет сообщать имя вызываемого метода и ничего более. Позднее это поможет нам определить, какие методы дерева наследования вызываются.

```
class Employee
{
  public void CalculatePay()
  {
    Console.WriteLine( "Employee.CalculatePay() ") ;
  }
}
```

А теперь допустим, что вы хотите создать класс, производный от Employee, и подменить метод CalculatePay, чтобы выполнять какие-либо действия, специфичные для производного класса. Для этого вам понадобится ключевое слово new с определением метода производного класса. Вот как это делается:

```
using System;
class Employee
{
 public void CalculatePay()
 {
 Console.WriteLine("Employee.CalculatePay()");
 }
}
class SalariedEmployee: Employee
{
 // Ключевое Слово new позволяет заменить

 // реализацию, содержащуюся в базовом классе.

 new public void CalculatePay()

 {console. WriteLine("SalariedEmployee.CalculatePay()");
}
class Poly1App
{
 public static void Main()
```

```
 {
 PolylApp poly1 = new PolylApp ();
 Employee baseE = new Employee ();
 baseE.CalculatePay f);
 SalariedEmployee S = new SalariedEmployee ();
 s.CalculatePay();
 }
}
```

Скомпилировав и запустив это приложение, вы получите такую информацию:

Employee.CalculatePay ()
Salaried. CalculatePay ()

ПОЛИМОРФИЗМ

Подмена методов с помощью ключевого слова new замечательно работает, если у вас есть ссылка на производный объект. А что будет, если у вас есть ссылка, приведенная к базовому классу, но нужно, чтобы компилятор вызывал реализацию метода из производного класса? Это именно тот момент, когда в дело вступает полиморфизм. Полиморфизм позволяет вам многократно определять методы в иерархии классов так, что в период выполнения в зависимости от того, какой именно объект используется, вызывается соответствующая версия данного метода.

Обратимся к нашему примеру с сотрудником. Приложение PolylApp работает корректно, поскольку у нас есть два объекта: Employee и SalariedEmployee. В более практичном приложении мы, вероятно, считывали бы все записи о сотрудниках из БД и заполняли ими массив. Хотя некоторые сотрудники работают по контракту, а другим выплачивается зарплата, все они должны быть помещены в массив в виде объектов одного типа — базового класса Employee. Но при циклической обработке массива, когда происходит получение каждого объекта и вызов его метода CalculatePay, нам бы хотелось, чтобы компилятор вызывал подходящую реализацию метода CalculatePay.

В следующем примере добавлен новый класс — ContractEmployee. Главный класс приложения теперь содержит массив объектов типа Employee и два дополнительных метода: LoadEmployees, загружающий объекты «сотрудник» в массив, и DoPayroll, обрабатывающий массив, вызывая метод CalculatePay для каждого объекта.

```
using System;

class Employee
f
  public Employee(string name)
```

```
  {
    this.Name = name;
  }
  protected string Name;
  public string name
  {
    get
    {
      return this.Name;
    }
  }

  public void CalculatePay()
  {
    Console.WriteLine("Employee.CalculatePay вызвана для {0}", name);
  }
}

class ContractEmployee: Employee
{
  public ContractEmployee(string name)
    : base(name)
  {
  }

  public new void CalculatePay()
  {
    Console.WriteLine("ContractEmployee.CalculatePay вызвана для {0}", name);
  }
}

class SalariedEmployee: Employee
{
  public SalariedEmployee (string name)
    : base(name)
  {
  }

  public new void CalculatePay()
  {
    Console.WriteLine("SalariedEmployee.CalculatePay вызвана для {0}", name);
  }
}

class Poly2App
```

```
{
  protected Employee[] employees;

  public void LoadEmployees()
  {
    // эмулируем загрузку из базы данных
    employees = new Employee[2];
    employees[0] = new ContractEmployee("Алексей Рубинов");
    employees[1] = new SalariedEmployee("Василий Лазерко");
  }

  public void DoPayroll()
  {
    foreach(Employee emp in employees)
    {
      emp.CalculatePay();
    }
  }

  public static void Main()
  {
    Poly2App poly2 = new Poly2App();
    poly2.LoadEmployees();
    poly2.DoPayroll();
  }
}
```

Однако, запустив это приложение (Poly2App), вы получите такую информацию:

Employee.CalculatePay вызвана для Алексей Рубинов
Employee.CalculatePay вызвана для Василий Лазерко

Ясно, что это совсем не то, что нам нужно: для каждого объекта вызывается реализация метода CalculatePay из базового класса. То, что здесь происходит,— пример раннего связывания (early binding). Во время компиляции этого кода компилятор C#, изучив вызов emp.CalculatePay, определил адрес памяти, на который ему нужно перейти во время этого вызова. В таком случае это будет адрес метода Employee.CalculatePay. Вызов метода Employee.CalculatePay представляет собой проблему. Мы хотим, чтобы вместо раннего связывания происходило динамическое (позднее) связывание (late binding). Динамическое связывание означает, что компилятор не выбирает метод для исполнения до периода выполнения. Чтобы заставить компилятор выбрать правильную версию метода объекта, мы используем два новых ключевых слова: virtual и override. С методом базового класса применяется ключевое слово virtual, а override —с реализацией метода в производном классе. Вот еще один пример, который на этот раз работает должным образом,— Poly3App.

```
using System;

class Employee
{
  public Employee(string name)
  {
    this.Name = name;
  }

  protected string Name;
  public string name
  {
    get
    {
      return this.Name;
    }
  }

  virtual public void CalculatePay()
  {
    Console.WriteLine("Employee.CalculatePay вызвана для {0}", name);
  }
}

class ContractEmployee: Employee
{
  public ContractEmployee(string name)
    : base(name)
  {
  }

  override public void CalculatePay()
  {
    Console.WriteLine("ContractEmployee.CalculatePay вызвана для {0}", name);
  }
}

class SalariedEmployee: Employee
{
  public SalariedEmployee (string name)
    : base(name)
  {
  }

  override public void CalculatePay()
```

```
  {
    Console.WriteLine("SalariedEmployee.CalculatePay вызвана для {0}", name);
  }
}

class Poly2App
{
  protected Employee[] employees;

  public void LoadEmployees ()
  {
    // эмулируем загрузку из базы данных
    employees = new Employee[2];
    employees[0] = new ContractEmployee("Алексей Рубинов");
    employees[1] = new SalariedEmployee("Василий Лазерко");
  }

  public: void DoPayroll()
  {
    foreach(Employee emp in employees)
    {
      emp.CalculatePay();
    }
  }

  public static void Main()
  {
    Poly2App poly2 = new Poly2App();
    poly2.LoadEmployees();
    poly2.DoPayroll();
  }
}
```

Выполнение такого кода приведет к следующим результатам:
ContractEmployee.CalculatePay вызвана для Алексей Рубинов
SalariedEmployee.CalculatePay вызвана для Василий Лазерко

СТАТИЧЕСКИЕ МЕТОДЫ

Статическим называется метод, который существует в классе как таковом, а не в отдельных его экземплярах. Как и в случае других статических членов, главное преимущество статических методов в том, что они расположены вне конкретных экземпляров класса и не засоряют глобальное пространство приложения. При этом они и не нарушают принципов ООП, поскольку ассоциированы с определенным классом. Примером может служить API баз данных, представленный ниже. В

той иерархии классов есть класс SQLServerDb. Помимо базовых операций для работы с БД (new, update, read и delete), класс содержит метод, предназначенный для восстановления БД. В методе Repair не нужно открывать саму БД. Следует использовать функцию ODBC SQLConfigDataSource, которая предполагает, что БД закрыта. Однако конструктор SQLServerDb открыл БД, указанную переданным ему именем. Поэтому здесь очень удобно использовать статический метод. Это позволит поместить метод в класс SQLServerDb, к которому он принадлежит, и даже не обращаться к конструктору класса. Очевидно, выгода клиента в том, что он также не должен создавать экземпляр класса SQLServerDb. В следующем примере вы можете видеть вызов статического метода (RepairDatabase) из метода Main. Заметьте, что для этого не создается экземпляр SQLServerDB:

```
using System;

class SQLServerDb
{
  public static void RepairDatabase()
  {
    Console.WriteLine("восстановление базы данных... " ] ;
  }
}

class StaticMethod1App
{
  public static void Main()
  {
  } SQLServerDb.RepairDatabase() ;
}
```

Определить метод как статический позволяет ключевое слово static. Затем для вызова метода пользователь применяет синтаксис вида Класс.Метод. Этот синтаксис необходим, даже если у пользователя есть ссылка на экземпляр класса. Данный момент можно проиллюстрировать кодом, который не будет компилироваться:

```
using System;

class SQLServerDb
{
  public static void RepairDatabase()
  {
    Console.WriteLine("восстановление базы данных... " ) ;
  }
}

class StaticMethod2App
```

```
{
  public static void Main()
  {
    SQLServerDb db = new SQLServerDb();
    db.RepairDatabase();
  }
}
```

Последним моментом, касающимся статических методов, является правило, определяющее, к каким членам класса можно обращаться из статического метода. Как вы можете догадаться, статический метод может обращаться к любому статическому члену в пределах класса, но не может обращаться к члену экземпляра. Например:

```
using System;

class SQLServerDb
{
  static string progressString1 = "repairing database...";
  string progressString2 = "repairing database...";

  public static void RepairDatabase()
  {
    Console.WriteLine(progressString1); // все правильно
    Console.WriteLine(progressString2); // не будет компилироваться
    // compilation.
  }

}

class StaticMethod3App
{
  public static void Main()
  t
  } SQLServerDb.RepairDatabase();
]
```

Компилятор не пропустит этот код, потому что он содержит ошибку. Статический метод пытается обратиться к нестатической переменной класса. Это недопустимо в C#.

РЕКУРСИЯ

Иногда возникают ситуации, когда в теле функции необходимо вызвать экземпляр этой же функции. Такое явление называется рекурсией. Использование рекурсии позволит вам в некоторых случаях избежать сложных циклов и проверок. Типичным примером использования рекурсии является функция нахождения факториала числа.

```
using System;
class Degrees
{
 public static int Factorial(int x)
 {
 return (x ==1) ? 1: x * Factorial(x-1);
 }
 public static void Main()
 {
 for (int i = 1; i < 10; i++)
 i
 Console.WriteLine(Factorial(i) ) ;
 }
 }
}
```

Функция Factorial является рекурсивной. На первый взгляд она может показаться вам сложной. Однако она совсем проста (всего одна строка). Если функция приняла значение, отличное от единицы, то вызывает саму себя и передает значение, на единицу меньшее. Так происходит до тех пор, пока переданое значение не станет равным единице. После того как функция примет значение 1, начнется выход из всех экземпляров вызванных функций Factorial и возврат произведения параметра функции и возвращенного значения.

Допустим, мы ищем факториал числа 5. Тогда передача параметров экземплярам функции Factorial идет в такой последовательности: 5, 4, 3, 2, 1. А последовательность возвращаемых значений будет следующей: 1, 2, 6, 24, 120. Результатом работы функции станет число 120.

Однако рекурсия имеет и недостатки. При создании каждого экземпляра функции выделяются дополнительные блоки памяти, что нежелательно при работе программы.

11. СВОЙСТВА

ПРИМЕНЕНИЕ СВОЙСТВ

C# поддерживает такой тип данных, как свойства. Свойства позволяют не только скрывать классам реализацию методов, но и запрещать членам любой прямой доступ к полям класса. Для обеспечения корректной работы с полями класса в C# введено такое понятие, как свойство метода. Свойство выполняет работу по получению и установке значений полей, действуя согласно прописанным требованиям.

Для примера рассмотрим класс, имеющий два поля: возраст и год рождения. Поле возраст хранит число, определяющее количество полных лет человека. Поле год рождения, соответственно, хранит год рождения человека. Как вы понимаете, эти поля связаны между собой очень жестко. Если предоставить программисту открытый доступ к полю возраст, то может возникнуть рассогласование данных в ходе работы программы, потому что изменение поля возраст не изменит значения поля год рождения.

Рассмотрим пример описанной выше программы:

```
using System;

namespace Properties
  class People
  {
    public int birthday;
    public int age;

    public People (int age)
    t
      this.age = age;
      this.birthday = 2003-age;
    }
  }

  class PeopleApp

   {
    static void Main(string[] args)
     (
      People pep = new People(22);
      pep.age = 30;
```

```
      Console.WriteLine("Возраст: {0}, Год рождения: {1}",
              pep.age, pep.birthday);
    }
  }
}
```

Результат работы программы будет следующий:

Возраст: 30, Год рождения: 1981

Это отнюдь не правильный результат. Если сейчас 2003 год, то при возрасте сотрудника в 30 лет его год рождения должен быть 1973. Однако значение birthday не изменилось после изменения значения age.

Один из способов избежать подобных недочетов — использование методов-аксессоров. Методы-аксессоры позволяют закрепить за каждым полем функцию, которая будет устанавливать значение поля или возвращать данные о значении поля.

Вот как выглядит усовершенствованный вариант предыдущего примера с использованием методов-аксессоров:

```
using System;
namespace Properties
{
  class People
  {
    private int birthday;
    private int age;
    public void SetAge(int age)
    {
      this.age = age;
      this.birthday = 2003 - age;
    }
    public int GetAge()
    {
      return this.age;
    }
    public void SetBirthday(int birthday)
    {
      this.birthday = birthday;
      this.age = 2003 - birthday;
    }
    public int GetBirthday()
    {
      return this.birthday;
    }

    public People(int age)
```

```
    {
      this.age = age;
      this.birthday = 2003-age;
    }
  }
  class PeopleApp
  {
    static void Main(string[] args)
    {
      People pep = new People(22);
      pep.SetAge(30);
      Console.WriteLine("Возраст: {0}, Год рождения: {1}",
        pep.GetAge(), pep.GetBirthday());
    }
  }
}
```

Результат работы программы будет следующий:

Возраст: 30, Год рождения: 1973

В этом примере для каждого поля определены функции Get и Set. Они предназначены для управления доступом к полям age и birthday, которые в данном случае являются скрытыми для внешнего использования. Функции GetAge и GetBirthday просто возвращают значения соответствующих полей. Функции SetAge и SetBirthday устанавливают значения соответствующих переменных и, кроме того, производят дополнительные действия по уравновешиванию зависимых переменных. Если вызывается функция SetAge, то значение age изменяется напрямую, а значение birthday изменяется по формуле 2003 — age. Такое же действие происходит и при вызове функции SetBirthday.

Однако использование функций-аксессоров вызывает некоторые неудобства. Функции-аксессоры имеют различные имена, заставляют использовать передачу параметров и анализ возвращаемого значения. Применение свойств помогает избежать этих неудобств. Свойства позволяют обращаться к членам класса так, как будто вы обращаетесь к полям. И вам даже не приходится задумываться, есть ли эти поля на самом деле. Вот как будет выглядеть наш пример с использованием свойств:

```
using System;

namespace Properties
{
  class People
  {
    private int birthday;
    private int age;

    public int Age
```

```
    {
      set
      {
        age = value;
        birthday = 2003 -value;
      }
      get
      {
        return age;
      }
    }

    public int Birthday
    t
      set
      {
        birthday = value;
        age = 2003 -value;
      )
      get
      {
        return birthday;
      }
    }

    public People(int age)
    {
      this.age = age;
      this.birthday = 2003 - age;
    }
  }
  class PeopleApp
  {
    static void Main(string[] args)
    {
      People pep = new People(22);
      pep.Age = 30;

      Console.WriteLine("Возраст: {0}, Год рождения: {1}",
        pep.Age, pep.Birthday);
    }
  }
]
```

Свойство на C# состоит из объявления поля и методов-аксесоров, применяемых для изменения значения поля. Эти методы-аксессоры называются получатель (getter) и установщик (setter). Методы-получатели используются для получения значения поля, а установщики — для его из-

менения. В нашем примере создано поле People.age и свойство People.Age. Не стоит думать, что совпадение имен имеет какое-то значение — нет. Разница между полями в регистре заглавной буквы призвана лишь улучшить читабельность кода. People.Age — это не поле, а свойство, представляющее собой универсальное средство определения аксессоров для членов класса, что позволяет использовать более интуитивно понятный синтаксис вида объект.поле. Свойство имеет два стандартных метода-аксессора: get и set. Заметьте также, что свойство не принимает значения аргумента. Аргумент имеет по умолчанию имя value.

На самом деле компилятор преобразует определение вашего свойства в вызовы функций-аксессоров. То, какой из методов-аксессоров будет вызываться, определяется местоположением свойства. Если свойство используется слева от оператора присваивания, то вызывается функция установщик:

```
pep.Age = 30;
```

Если же свойство используется для получения значения, то вызывается функция получатель:

```
Console.WriteLine("Возраст: {0}, Год рождения: {1}",
  pep.Age, pep.Birthday);
```

СВОЙСТВА ТОЛЬКО ДЛЯ ЧТЕНИЯ

В нашем примере оба свойства считаются доступными и для чтения, и для записи. Однако вы можете определить свойство как доступное только для чтения. Определять свойство только для записи не имеет смысла, поскольку им нельзя будет воспользоваться. Для того чтобы определить свойство как доступное только для чтения, вам необходимо определить для него только функцию получатель. Вот как можно сделать для нашего примера свойство Birthday доступным только для чтения:

```
using System;

namespace Properties
{
  class People
  {
    private int birthday;
    private int age;

    public int Age
    {
      set
      {
        age = value;
        birthday = 2003 - value;
      }
      get
```

```
        {
            return age;
        }
    }

    public int Birthday
    {
        get
        {
            return birthday;
        }
    }

    public People(int age)
    {
        this.age = age;
        this.birthday = 2003-age;
    }
}
class PeopleApp
{
    static void Main(string[] args)
    {
        People pep = new People(22);
        pep.Age = 30;

        //pep.Birthday = 1800;

        Console.WriteLine("Возраст: {0}, Год рождения: {1}",
            pep.Age, pep.Birthday);
    }
}
```

Здесь свойство People.Birthday имеет только функцию-получатель. Если вы попробуете использовать в функции Main код для изменения свойства People.Birthday, то программа не скомпилируется. Раскомментируйте в программе строку

```
//pep.Birthday = 1800;
```

Попробуйте скомпилировать программу. Компилятор выдаст ошибку с сообщением, что данное свойство доступно только для чтения.

СВОЙСТВА И НАСЛЕДОВАНИЕ

Свойства, как и методы, могут быть перегружены в производных классах. Для свойств также могут задаваться модификаторы virtual, override или abstract. Подробно о наследовании говорилось в главе «Методы», сей-

час я лишь хочу еще раз привести подробный пример использования механизма виртуализации с применением свойств:

```
using System;

enum COLORS
{
  RED,
  GREEN,
  BLUE
}

namespace Properties
{
  abstract class Base
  {
    protected COLORS color;

    public abstract COLORS Color
    {
      get;
      set;
    }

    protected int size;
    public virtual int Size
    {
      set
      {
        size = value;

        //изменение размера объекта
        Console.WriteLine("Изменение размера объекта {0}", value);
      }
      get
      {
        return size;
      }
    }
  }

  class Circle: Base
  {
    public override COLORS Color
    {
      get
      {
        return color;
      }
```

```
    set
    {
      color = value;

      //код для изменения цвета окружности
      Console.WriteLine("Изменение цвета окружности {0}", color.ToString());
    }
  }

  private int radius;
  public int Radius
  {
    get
    {
      return radius;
    }
  }

  public override int Size
  {
    get:
    (
      return size;
    }
    set
    {
      size = value;
      radius = value;

      //код для перерисовки окружности с новым размером
      Console.WriteLine("Изменение размера (радиуса) окружности {0}", value);
    }
  }
}

class Bar: Base
{
  public override COLORS Color
  {
    get
    1
      return color;
    }
    set
    \
      color = value;
```

```
        //код для изменения цвета квадрата
        Console.WriteLine("Изменение цвета квадрата {0}", value.ToString());
      }
    }
  }

class PeopleApp
{
  static void Main(string[] args)
  {
    Base baseObj;

    Circle circle = new Circle();
    Bar bar = new Bar();

    //работаем с объектом Circle через базовый класс
    baseObj = circle;

    //будет использовано перегруженное свойство класса Circle
    circle.Color = COLORS.BLUE;

    //будет использовано перегруженное свойство класса Circle
    circle.Size = 10;

    //работаем с объектом Bar через базовый класс
    baseObj = bar;

    //будет использовано перегруженное свойство класса Bar
    baseObj.Color = COLORS.GREEN;

    //будет использовано свойство базового класса
    baseObj.Size = 50;

  }
}
```

Условно говоря, данное приложение предназначено для рисования окружностей и квадратов. Для наглядности объекты могут иметь только три цвета, которые определены перечислением COLORS.

```
enum COLORS
{
  RED,
  GREEN,
  BLUE
}
```

В качестве основы для построения объектов взят абстрактный базовый класс Base.

```
abstract class Base
{
  protected COLORS color;
  public abstract COLORS Color
  (
    get;
    set;
  }
  protected int size;
  public virtual int Size
  (
    set { ... }
    get { ... }
  }
}
```

Этот класс имеет два защищенных поля color и size. Для доступа к данным полям используются свойства Color и Size.

Свойство Color объявлено с модификатором abstract. Это означает, что в базовом классе нет реализации данного свойства и все производные классы обязательно должны перегрузить это свойство. О том, что свойство Color в базовом классе не имеет реализации, свидетельствуют пустые функции установщик и получатель.

Свойство Size объявлено со спецификатором virtual. Это означает, что производные классы могут использовать данное свойство как в варианте базового класса, так и определить свой вариант реализации свойства. Поэтому свойство Size имеет реализацию в классе Base.

Код программы содержит два производных класса от класса Base: Circle и Bar.

```
class Circle: Base f ... j
class Bar: Base { . . . }
```

Поскольку класс Base является абстрактным и содержит абстрактное свойство Color, то оба производных класса обязаны реализовать у себя свойство Color. Они перегружают свойство Color, устанавливая не только значение цвета, но и реализуя код, для раскраски реальных объектов. Заметьте, что свойство Color имеет тип перечисления COLORS. Поэтому переменная value, видимая внутри метода-установщика или метода-получателя, имеет тип перечисления COLORS. О том, что мы перегружаем абстрактное свойство базового класса, свидетельствует ключевое слово override, используемое при объявлении свойств в производных классах.

```
public override COLORS Color
```

Свойство Size не является абстрактным, поэтому производные классы могут не перегружать его. Именно так и сделано в классе Bar. Он не содержит перегруженного свойства Size.

В свою очередь класс Circle содержит перегруженное свойство Size. Оно перегружено таким образом, что, кроме установки значения поля size базового класса, устанавливает значение поля radius.

```
public override int Size
{
  get { ... }
  set
  {
    size = value;
    radius = value;
      //код для перерисовки окружности с новым размером
    Console.WriteLine("Изменение размера (радиуса) окружности {0}", value);
  }
}
```

Класс PeopleApp является своего рода клиентом, использующим созданные нами классы.

В первой строке функции Main объявляется объект класса Base. Заметьте, именно объявляется, а не создается. Объект абстрактного класса создать нельзя.

```
Base baseObj;
```

Далее формируются два объекта производных классов.

```
Circle circle = new Circled;
Bar bar = new Bar();
```

Для наглядной демонстрации механизма виртуализации будем работать со свойствами производных классов через объект базового класса.

Первым делом присваиваем переменной baseObj объект circle. Как уже говорилось, объект производного класса может быть приведен к типу базового класса, но не наоборот. При этом все виртуальные свойства и методы базового класса заменятся перегруженными свойствами и методами производного класса.

```
baseObj = circle;
```

Объект circle имеет тип Circle, который содержит два перегруженных свойства Color и Size. Когда мы будем устанавливать значения этих свойств, то будет происходить обращение к экземплярам свойств из производного класса.

```
circle.Color = COLORS.BLUE;
circle.Size = 10;
```

При выполнении данных строк кода на экране появится:

Изменение цвета окружности BLUE

Изменение размера (радиуса) окружности 10

После этого переменной baseObj присваивается объект bar, имеющий тип класса Bar.

```
baseObj = bar;
```

Теперь baseObj при обращении к свойству Color будет использовать экземпляр свойства Color класса Bar. Со свойством Size дело обстоит иначе. Класс Bar не имеет перегруженного экземпляра свойства Size. Поэто-

му, при обращении к свойству Size, обращение будет происходить к свойству базового класса.

```
baseObj.Color = COLORS.GREEN;
baseObj.Size = 50;
```

При выполнении этих строк кода на экран выведется:
Изменение цвета квадрата GREEN
Изменение размера объекта 50

ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ СВОЙСТВ

Одним из преимуществ использования свойств является реализация методики отложенной инициализации. При этой методике некоторые члены класса не инициализируются, пока не потребуется.

Отложенная инициализация дает преимущества, если у вас имеется класс с членами, на которые редко ссылаются и на инициализацию которых уходит много времени и ресурсов. Типичным примером могут служить ситуации, в которых требуется считать значения из базы данных или получить данные по сети. Если вам известно, что на эти члены ссылаются редко, а на их инициализацию требуется много ресурсов, то инициализацию полей можно отложить до вызова методов получателей.

Для примера давайте рассмотрим класс, имеющий свойство, значение которого необходимо получать из базы данных. Допустим, у вас имеется многопользовательская система. При поставке этой системы заказчику вы продаете определенное количество лицензий. Количество лицензий означает количество экземпляров программы, которые могут быть одновременно запущены. Данные о количестве уже запущенных экземпляров программы хранятся на удаленном сервере в базе данных. Получить это значение возможно, лишь затратив большое количество ресурсов. Вашему классу необходима проверка лицензии только в редких случаях. Поэтому нет необходимости включать код для получения значения о количестве лицензий в конструктор класса. Проще включить код для инициализации поля в функцию получатель соответствующего свойства.

```
class Customer
{
  private int licenseCount;
  public int LicenseCount
  {
    get
    {
      // получить значение количества используемых
      // лицензий с удаленного сервера и записать
      // их в переменную licenseCount

      return licenseCount;
    }
  }
}
```

12. МАССИВЫ

В главе 4 я уже рассказывал о применении массивов как типов данных. Но понятия массивов в C# распространяются гораздо шире, нежели просто объединение множества значений в переменной с одним именем.

В C# массивы являются объектами, производными от базового класса System.Array. Поэтому, несмотря на синтаксис определения массивов, аналогичный языку C++, на самом деле происходит создание экземпляра класса, унаследованного от System.Array.

ОДНОМЕРНЫЕ МАССИВЫ

Если вы объявляете массив как

```
int[]  arr;
```

то объявляете объект класса, производный от System.Array.

Как уже отмечалось, для создания объекта массива необходимо использовать оператор new. Приведу полный пример приложения, работающего с массивами.

```
using System;

namespace C_Sharprogramming
{
  class ArrClass
  (
    public int[] arr;
    public ArrClass()
    {
      arr = new int [5];
      for(int i = 0; i < 5; i++)
      {
        arr[i] = i;
      }
    }
  }

  class ArrApp
  i
    static void Main(string[] args)
    {
      ArrClass arrObj = new ArrClass ();
```

```
        for(int i = 0; i < 5; i++)
        {
          Console.WriteLine("arr[{0}]: = {1}", i, arrObj.arr[i]);
        ]
      }
    }
}
```

Класс ArrClass содержит объявление массива arr. В конструкторе класса массив arr инициализируется пятью элементами.

Функция Main создает экземпляр класса ArrClass. При создании экземпляра класса вызывается конструктор ArrClass, который инициализирует массив. После инициализации функция Main выводит значения элементов массива на экран.

arr[0]: =O
arr[1]:= 1
arr[2]: = 2
arr[3]: =3
arr[4]: = 4

МНОГОМЕРНЫЕ МАССИВЫ

О многомерных массивах я уже рассказывал в главе 4. Но далеко не все было упомянуто. Начнем с того, что представляет собой многомерный массив.

Многомерный массив — это массив, элементами которого являются массивы. Если массив имеет порядок N, то элементами такого массива являются массивы порядка N-1. Порядок еще называется рангом массива. Массив может иметь любой ранг, хотя вряд ли вам пригодится массив ранга 10, обычно используют одно-, двухранговые (двумерные) массивы. В некоторых случаях применяются массивы с рангом три.

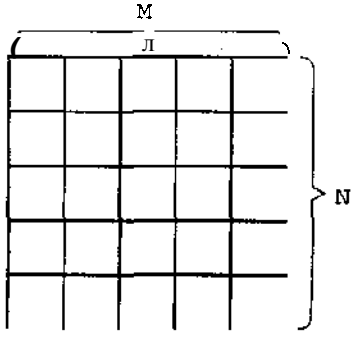


Рис. 12.1. Двумерный массив

Рассмотрим двумерный массив.

Двумерный массив, представленный на рис. 12.1, состоит из N строк и M столбцов. Значит, его размерность составляет N на M. Если вы захотите объявить такой массив, то это необходимо сделать следующим образом:

```
int[,] arr = new inc[N, M];
```

Каждая из N строк содержит по M элементов (столбцов). При обращении к элементу массива следует указать номер строки и номер столбца.

```
int n = arr[N1, M1]
```

Таким же способом можно работать с трех-, четерых-, N-мерными массивами. Не утруж-

дайте себя представлением N-мерного пространства. Это ни к чему. Лучше представьте себе использование многомерного массива. Например, необходимо разработать программу, которая использует следующую модель данных: имеется ограниченное число университетов, в каждом университете существует несколько факультетов, на каждом факультете студенты обучаются несколько курсов. Вам необходимо хранить информацию обо всех студентах. Как вы это сделаете? Сразу хочу отметить, что и количество университетов, и количество факультетов, и количество курсов являются известными строго ограниченными значениями, потому как массивы используются для статического хранения данных. Вы не можете в любой момент изменить размерность массива, и в этом заключается самый большой недостаток массивов. Именно из-за того, что размерность массива должна быть заранее известна, массивы используются лишь в простейших случаях хранения данных. Для динамического хранения данных предназначены списки, но об этом немного позже.

Рассмотрим далее наш пример. У вас имеется N университетов, M факультетов и K курсов, на каждом курсе учится S студентов. Массив какой размерности вам придется использовать для хранения этих данных? Ответ однозначный — четырехмерный. То есть, мы объявим наш массив как:

```
string arr [,,,];
arr = new string [N,M, K, S] ;
```

Каждая ячейка массива arr[N] является трехмерным массивом. Значит, если мы знаем индекс университета в массиве, то, написав arr[idx] (где idx — это индекс университета), конкретизируемся с выбором нужного трехмерного массива. Указав индекс второго порядка arr[idx][idx2j, выбираем в обозначенном трехмерном массиве двумерный массив со студентами всех курсов данного факультета. Указав индекс третьего порядка, можно выбрать необходимый вам курс факультета arr[idx][idx2][idx3]. Для выбора конкретного студента используйте индекс четверого порядка arr[idx][idx2][idx3][idx4]. Это на первый взгляд сложное описание является простой задачей.

Давайте рассмотрим пример использования многомерного массива. Возьмем двумерный массив строк для хранения данных.

```
using System;

namespace Multidimensional
t
  class MainApp
  (
    static void Main(string[] args)
    {
      string[,] arr;
      const int firstIdx = 2;
      const int secondIdx = 3;

      //создаем массив
```

```
      arr = new string[firstIdx, secondIdx];

      // заполняем двумерный массив
      for(int i = 0; i < firstIdx; i++)
        for (int j = 0; j < secondIdx; j++)
        {
          arr[i,j] = "arr " + (i) + "-" + (j);
        }

      // выводим значения из массива
      for(int i = 0; i < firstIdx; i++)
      {
        for (int j = 0; j < secondIdx; j++)
        {
          Console.WriteLine("Поле " + i + "-" + j);
        }
      }
    }
  }
}
```

Двумерный массив объявляется с использованием одной разделительной запятой внутри квадратных скобок.

```
string[,] arr;
```

Объявляем две переменные, которые будут хранить размерность массива.

```
const int firstIdx = 2;
const int secondIdx = 3;
```

В данном случае размерность массива будет составлять 2*3. Далее следует создание массива.

```
arr = new string[firstIdx, secondIdx];
```

Создание массива тесно связано с его объявлением. Нельзя объявить двумерный массив, а при создании указать три параметра размерности. Теперь массив создан, под него выделена память, и он может быть заполнен данными. Следующий код заполняет массив строками, которые хранят значения номеров строк и столбцов элементов.

```
for (int i = 0; i < firstIdx; i++)
  for (int j = 0; j < secondIdx; j++)
  {
    arr[i, j] = "arr " + (i) + "-" + (j);
  }
```

Так же, как и заполнение массива, выполняется вывод его значений на экран.

```
for(int i = 0; i < firstIdx; i++)
{
  for(int j = 0; j < secondIdx; j++)
  {
    Console.WriteLine("Поле " + i + "-" + j);
  }
}
```

Результат работы программы будет следующий:

Поле0-0
Поле0-1
Поле0-2
Поле1-0
Поле1-1
Поле1-2

РАЗМЕР И РАНГ МАССИВА

Как уже говорилось, любой массив является объектом класса, производным от System.Array. Как и любой другой класс, System.Array имеет свои методы и свойства. Одним из важнейших методов класса System.Array является метод GetLenght(). Он позволяет получить размер определенного измерения многомерного массива. Другое важное достоинство класса System.Array — свойство Rank, которое позволяет программным путем определить ранг массива, то есть количество измерений массива. Кроме того, класс System.Array имеет свойство Length, которое возвращает общее количество элементов в массиве. Давайте рассмотрим использование этих возможностей, видоизменив предыдущий пример.

```
using System;

namespace Multidimensional
{
  class MainApp
  {
    static void Main(string[] args)
    {
      string[,] arr;
      const int firstIdx = 2;
      const int secondIdx = 3;

      //создаем массив
      arr = new string[firstIdx, secondIdx];

      // получаем значение ранга массива
      int Rank = arr.Rank;
      Console.WriteLine("Массив arr имеет ранг {0}", Rank);

      // определяем общее количество элементов массива
      int Length = arr.Length;
      Console.WriteLine("Количество элементов в массиве: {0}", Length);

      // заполняем двумерный массив
```

```
      for(int i = 0; i < arr.GetLength(0); i++)
        for(int j = 0; j < arr.GetLength(1); j++)
        {
          arr[i,j] = "arr " + (i) + "-" + (j);
        }

      // выводим значения из массива
      for(int i = 0; i < arr.GetLength(0) ; i++)
        for(int j = 0; j < arr.GetLength(1) ; j++)
        {
          Console.WriteLine("Поле " + i + "-" + j);
        }
    }
  }
}
```

В этом примере вычисляется ранг массива и выводится его значение на экран. При обходе элементов массива нет необходимости запоминать размер каждого измерения массива. Размер соответствующего измерения можно получить при помощи функции Getl_ength(). Результат работы программы:

Массив arr имеет ранг 2
Количество элементов в массиве: 6
Поле0-0
Поле0-1
Поле0-2
Поле1-0
Поле1-1
Поле1-2

НЕВЫРОВНЕННЫЕ МАССИВЫ

Многомерные массивы могут иметь как одинаковый размер всех вложенных массивов, так и различный. Если используемый в программе массив будет иметь различную длину вложенных массивов, то об этом следует позаботиться заранее. Объявлять такой массив следует как:

```
int[][] arr;
```

Такое объявление будет означать, что объявляется массив массивов значений типа int. Приведу пример использования невыровненного массива. Допустим, у вас есть массив для хранения фамилий мальчиков и девочек в учебном классе. Для большей управляемости, вам лучше разделить мальчиков и девочек на различные массивы. Объявив невыровненный массив, вы можете создать родительский массив, что позволит с легкостью обрабатывать всех учеников в классе.

```
using System;

namespace Multidimensional
```

```
{
  class Pupil
  {
    virtual public void Hello()
    {
      Console.WriteLine("Базовый класс");
    }
  }
  class Boy: Pupil
  {
    override public void Hello()
    {
      Console.WriteLine("Класс Boy");
    }
  }
  class Girl: Pupil
  {
    override public void Hello()
    {
      Console.WriteLine("Класс Girl");
    }
  }

  class MainApp
  {
    static void Main(string[] args)
    {
      Pupil[][] pupils;
      // создаем родительский массив
      pupils = new Pupil[2][];

      // создаем массив для мальчиков
      pupils[0] = new Pupil[5];
      // заполняем массив
      for(int i = 0; i < pupils[0].Length; i++)
      {
        pupils[0][i] = new Boy();
      }

      // создаем массив для девочек
      pupils[1] = new Pupil[8];
      // заполняем массив
      for(int i = 0; i < pupils[1].Length; i++)
      {
        pupils[1][i] = new Girl();
      }

      for(int i = 0; i < pupils.Length; i++)
```

```
    {
      forfint j = 0; j < pupils [i] .Length; j++)
      {
        Pupil pupil = pupils[i][j];
        Console.Write("{0}-{1}: ", i, j) ;
        pupil.Hello();
      }
    }
  }
}
J
```

Здесь я создал базовый класс Pupil. Он содержит всего одну виртуальную функцию Hello(), которая выводит на экран сообщение. Классы Boy и Girl являются производными от Pupil. Они перегружают функцию Hello и выводят собственное сообщение на экран.

В функции Main создается невыровненный массив объектов Pupil. При этом по нулевому индексу содержится массив для мальчиков, а по первому — массив для девочек. В итоге мы создали массив на 5 элементов для мальчиков и массив на 8 элементов для девочек. Каждый элемент массива был проинициализирован своим типом объекта. В итоге у нас в массиве оказалось 5 объектов типа Boy и 8 объектов типа Girl.

Для перебора всех элементов массива используются вложенные циклы. Поскольку массив pupils является невыровненным, то его свойство Length вернет количество элементов в первом измерении. Таким образом, цикл

```
forfint i = 0; i < pupils.Length; i++)
```

сделает 2 итерации. Вложенный цикл

```
forfint j = 0; j < pupils[i].Length; j++)
```

будет выполняться число раз, соответствующее количеству элементов в массиве. Первый раз он выполнится 5 раз, а второй — 8.

Благодаря полиморфизму, через базовый класс можно работать с производными классами. Что и делается с присвоением объекту pupil класса Pupil различных элементов массива. Результат работы программы будет следующий:

0-0: Класс Boy
0-1: Класс Boy
0-2: Класс Boy
0-3: Класс Boy
0-4: Класс Boy
1-0: Класс Girl
1-1: Класс Girl
1-2: Класс Girl
1-3: Класс Girl
1-4: Класс Girl
1-5: Класс Girl
1-6: Класс Girl
1-7: Класс Girl

ОПЕРАТОР foreach

В Visual Basic уже давно встроен оператор для итерирования массивов и коллекций. Поэтому разработчики C# сочли полезным наделить такой возможностью и язык C#. Синтаксис оператора foreach следующий:

```
foreach (тип in выражение)
```

Рассмотрим следующий код программы:

```
using System;
using System.Collections;

namespace ForEachApp
{
  class MyArray
  {
    public ArrayList peoples;

    public MyArray()
    {
      peoples = new ArrayList();
      peoples.Add("Иванов");
      peoples.Add("Петров");
      peoples.Add("Сидоров");
    }
    static void Main(string[] args)
    {
      MyArray arr = new MyArray();
      for(int i = 0; i < arr.peoples.Count; i++)
      {
        Console.WriteLine("{0}", arr.peoples[i]);
      }
    }
  }
}
```

Для перебора всех элементов массива используется цикл for. Такой способ наиболее прижился среди программистов на C++. Однако такой способ имеет ряд проблем: необходимость инициализации переменной цикла, проверки булева значения, наращивания переменной цикла, использования переменной определенного типа для выделения элемента массива. Использование оператора foreach позволяет избежать всех этих неудобств. Вот как можно переписать предыдущий пример с помощью оператора foreach.

```
foreach(string people in arr.peoples)
{
  Console.WriteLine("{0}", people);
}
```

Насколько проще и понятнее использование оператора foreach! Вам гарантирован обход всех элементов массива. Вам не нужно вручную устанавливать значение переменной для инициализации и проверять границы массива. Кроме того, foreach сам поместит необходимое значение в переменную указанного типа. Если вам нужно прервать цикл foreach, вы можете воспользоваться операторами break и return.

СОРТИРОВКА

Одной из наиболее распространенных операций при работе с массивами является сортировка. Существует множество способов сортировки, которые отличаются быстродействием и количеством используемой памяти. Я расскажу лишь о наиболее простом методе сортировки — сортировке пузырьком.

Задача сортировки заключается в следующем: имеется список целых чисел, необходимо упорядочить элементы в списке по возрастанию, то есть расположить их так, чтобы самый первый элемент был самым маленьким, второй больше первого, но меньше третьего и т.д.

При пузырьковой сортировке упорядоченный список получается из исходного путем систематического обмена пары рядом стоящих элементов, не отвечающих требуемому порядку; до тех пор, пока такие пары существуют.

Наиболее простой метод систематического обмена соседних элементов с неправильным порядком при просмотре всего списка слева направо определяет пузырьковую сортировку: максимальные элементы как бы всплывают в конце списка.

Пример:

```
B {20, -5, 10, 8, 7}, исходный список
B1{-5, 10, 8, 7, 20}, первый просмотр
B2{-5, 8, 7, 10, 20}, второй просмотр
B3{-5, 7, 8, 10, 20}, третий просмотр
```

При первом просмотре меняются местами 20 и –5, 20 и -10, 20 и 8, 20 и 7. При втором просмотре 10 и 8, 10 и 7. При третьем просмотре 8 и 7. В итоге самое большое число оказывается с правой стороны, а самое маленькое — с левой. Количество итераций n^2, где n — количество элементов в массиве. Ниже приведен пример программы, использующей пузырьковую сортировку.

```
using System;

namespace Sorting
{
  class TestApp
  {
    public static void Main()
    {
      int[] arr;
```

```
      arr = new int[10];

      Console.WriteLine("Исходный массив:");
      forlint i = 0; i < arr.Length; i++)
      (
        arr[i] = 10 - i;
        Console.Write("{0} ", arr[i]);
      }

      bool bSort = false;
      do
      {
        bSort = false;
        for(int i = 1; i < arr.Length; i++)
        {
          if (arr [i] < arr[i-1])
          {
            int c = arr[i];
            arr[i] =arr[i-1];
            arr[i-1] = c;
            bSort = true;
          }
        }
      } while(bSort);

      Console.WriteLine("Отсортированный массив:");
      for(int i = 0; i < arr.Length; i++)
      <
        Console.Write("{0} ", arr[i]);
      }

    }
  }
}
```

Здесь сортировка выполняется до тех пор, пока хотя бы одна пара элементов массива будет переставлена. О перестановке сигнализирует флаг bSort, который устанавливается в false при начале обхода массива и изменяется на true, если хотя бы одна пара элементов была переставлена.

13. ИНДЕКСАТОРЫ

В предыдущей главе подробно рассказывалось о том, как использовать массивы, которые представляют собой очень удобный способ хранения однотипной информации. В C# существует особая возможность, позволяющая работать с объектами так, как если бы они были массивами,— это механизм индексаторов.

ПРЕИМУЩЕСТВО ИСПОЛЬЗОВАНИЯ ИНДЕКСАТОРОВ

Вы можете спросить: «Зачем нужны индексаторы, в чем преимущество их использования?» Ответ такой: «Процесс написания приложений становится интуитивно более понятным». Индексаторы можно сравнить со свойствами. Последние позволяют обращаться к полям класса без использования методов аксессоров. При использовании свойств программисту нет необходимости задумываться о формате метода-установщика или метода-получателя. Программист выбирает простую инструкцию типа объект.поле. То же можно сказать и об индексаторах. Они позволяют обращаться к полям объекта так, как если бы это был массив.

Рассмотрим пример. Тем, кто хорошо знаком с программированием на Win32API, известно, каким образом происходит работа с элементом управления ListBox. Для того чтобы добавить в него строку, следует послать окну соответствующее сообщение с необходимыми параметрами. Этот метод имеет свои преимущества. Он поддерживается ядром Windows и является самым быстрым способом работы со списком. Однако со временем стали более широко использоваться такие языки программирования, как C++, Object Pascal и др., которые применяли интуитивно более понятные интерфейсы. Библиотека классов MFC использовала тонкую надстройку в виде классов, которые лишь отсылали соответствующие сообщения при работе с окнами. И программисты выбирали удобство использования программного кода, даже если приходилось жертвовать размером программы или ее быстродействием.

Все это было принято во внимание разработчиками языка C#. В стремлении разработать максимально интуитивно понятный язык программирования они задумались над вопросом возможности обработки объекта, который по своей сути является массивом, как массива. Ведь обычное окно со списком является тем же массивом с дополнительной возможностью отображения данных. В связи с этим и родилась концепция индексаторов.

Пример использования индексаторов можно найти и в языке C++. Там это называлось перегрузкой оператора ([ ]).

ОПРЕДЕЛЕНИЕ ИНДЕКСАТОРОВ

Как уже было сказано выше, индексаторы очень похожи на свойства. Формально синтаксис определения индексатора таков:

```
[атрибуты] [модификаторы] тип this [список-формальных-параметров]{set get}
```

Или, если это индексатор в интерфейсе, таков:

```
[атрибуты] [модификаторы] тип интерфейс. this [список формальных
параметров]{set get}
```

Где:

атрибуты — дополнительная информация об индексаторе. Наиболее значимым для индексаторов является атрибут Name. Задание Name позволяет дать имя индексатору для того, чтобы его могли использовать другие языки, не поддерживающие индексаторы. По умолчанию все индексаторы вашего класса имеют Name, равный Item;

модификаторы — модификаторы доступа и директивы. К индексатору применимы почти все стандартные директивы C#. Он может быть скрыт, перегружен, сделан виртуальным, но есть одно исключение, индексатор не может быть static;

список формальных параметров — указывает параметры, посредством которых осуществляется индексация. Передается в get и set, которые используются в индексаторе так же, как в свойствах. get применяется для вычисления индексатора по заданному списку формальных параметров, а set — для изменения индексатора. set получает в качестве дополнительного параметра value того же типа, что и индексатор.

Следует отметить, что доступ к индексатору осуществляется посредством сигнатуры, в отличие от свойств, доступ к которому осуществляется посредством имени. Сигнатурой индексатора считаются число и тип формальных параметров. Тип самого индексатора и имена параметров в сигнатуру не входят. Естественно, в классе не может быть двух индексаторов с одинаковой сигнатурой. К тому же, индексатор не считается переменной и не может быть передан в качестве ref или out параметра.

Вот пример простейшего класса, использующего индексатор.

```
class MyArray
{
  public object this [int idx]
  {
    get
    {
      // установка нужных значений
    }
    set
    {
      // возврат нужных значений
    }
  }
}
```

Здесь не приводится реализация индексатора. Здесь лишь описываются общие правила синтаксиса, используемого при создании индексаторов. По этому коду вы уже можете определить, что индексатор сродни вызову функции с переданным ей параметром и возвращающей значение. Однако индексаторы позволяют использовать в программе примеры кода подобно следующему:

```
MyArray obj = new MyArrayO;
obj[0] = someObj;
Console.WriteLine("{0}", obj[0]);
```

Клиентская часть программы не заботится о том, что возвращает индексатор. Все, что делается внутри индексатора, не имеет значения для клиентского кода. Клиент лишь ожидает от индексатора такого значения, как если бы он обращался к элементу массива.

Давайте рассмотрим пример использования индексатора:

```
using System;
using System.Collections;

namespace Indexers
{
  class ListBox
  {
    protected ArrayList data = new ArrayList();

    public object this [int idx]
    {
      get
      {
        // установка нужных значений
        if(idx > -1 &s idx < data.Count)
        {
          return data[idx];
        }
        else
        {
          // запрещенная ситуация
          // возникновение исключения
          return null;
        }
      }
      set
      {
        // установка нужных значений
        if(idx > -1 && idx < data.Count)
        {
          data[idx] = value;
        }
```

```
        else if(idx == data.Count)
        {
          data.Add(value);
        }
        else
        {
          // запрещенная ситуация
          // возникновение исключения
        }
      }
    }
  }

  class ListBoxApp
  {
    static void Main(string[] args)
    {
      ListBox list = new ListBox();
      list[0] = "красный";
      list[1] = "зеленый";
      list[2] = "синий";
      Console.WriteLine("{0} {1} {2}", list[0], list[1], list[2]);
    }
  }
}
```

Созданный нами класс ListBox имеет индексатор. Сам класс предназначен для хранения списка объектов. Класс ArrayList библиотеки классов .NET Framework предназначен для хранения совокупности объектов. Объект data типа ArrayList является членом класса ListBox. В моей реализации класса ListBox выполняется лишь проверка на выход индекса за границы допустимых значений и генерация исключения в случае необходимости. Вы можете расширить реализацию индексатора любыми дополнительными функциями. Например, добавить функцию отображения добавленного в список элемента на экране. От этого не изменится код клиентской части программы. Вам лишь необходимо будет изменить внутреннюю реализацию индексатора.

14. АТРИБУТЫ

Большинство языков программирования разрабатываются с учетом набора необходимых возможностей. Так, в начале создания компилятора вы думаете, какова будет структура приложений на новом языке, как один фрагмент кода будет вызывать другой, как распределить функциональность, и о многих других проблемах, решение которых сделает язык продуктивным средством создания ПО. Обычно разработчикам компиляторов приходится иметь дело со статическими сущностями. Например, вы определяете класс на C#, помещая перед его именем ключевое слово class. После этого вы определяете производный класс, вставляя двоеточие после его имени, за которым следует название базового класса. Это может служить примером решения, которое принимается разработчиком языка однажды и навсегда.

Сейчас те, кто пишет компиляторы, семь раз отмерят, прежде чем отрежут. Но даже они не могут предвидеть все будущие усовершенствования в нашей области и то, как они повлияют на способы выражения программистами типов на данном языке. Скажем, как создать связь между классом на C++ и URL документации для данного класса? Или как вы будете ассоциировать члены классов C++ с полями XML в новом решении вашей компании в области электронной коммерции? Поскольку C++ разрабатывался задолго до прихода в нашу жизнь Интернета и протоколов, таких как XML, обе эти задачи выполнить довольно трудно.

До сих пор решения подобных проблем предполагают хранение дополнительной информации в отдельном файле (DEF, IDL и т. д.), которая затем связывается с тем или иным типом или членом. Так как компилятор не обладает сведениями о каком-то файле или связи между вашим классом и этим файлом, такой подход обычно называется *разрывным решением* (disconnected solution). Главная проблема в том, что класс больше не является «самоописывающимся», т. е. теперь пользователь не может сказать о классе все, лишь взглянув на его определение. Одно из преимуществ самоописывающегося компонента — гарантия соблюдения при компиляции и в период выполнения правил, ассоциированных с компонентом. Кроме того, сопровождение самоописывающегося компонента проще, поскольку разработчик может найти всю связанную с ним информацию в одном месте.

Так все и было многие десятилетня. Разработчики языка пытаются определить, что вы хотите от языка, создают компилятор с этими возможностями, и, к счастью или к сожалению, вы остаетесь с этими возможностями до прихода следующего компилятора. Так это и было вплоть до сегодняшнего дня. C# предлагает иную парадигму, берущую начало от *атрибутов* (attributes).

НАЗНАЧЕНИЕ АТРИБУТОВ

Атрибуты предоставляют универсальные средства связи данных (в виде аннотаций) с типами, определенными на C#. Вы можете применять их для определения информации периода разработки (например, документации), периода выполнения (например, имя столбца БД) или даже характеристик поведения периода выполнения (например, может ли данный член участвовать в транзакции). Возможности атрибутов бесконечны.

Лучше объяснить использование атрибутов на примере. Допустим, у вас есть приложение, хранящее некоторые данные в реестре. Одна из проблем разработки связана с выбором места хранения информации о разделе реестра. В большинстве сред разработки она, как правило, хранится в файле ресурсов, в константах или даже жестко запрограммирована в вызовах API реестра. Однако мы снова имеем ситуацию, когда неотъемлемая часть класса хранится отдельно от определения остальной части класса. Атрибуты позволяют «прикреплять» эту информацию к членам класса, получая полностью самоописывающийся компонент. Вот пример, иллюстрирующий, как это может выглядеть, если предположить, что атрибут RegistryKey уже определен:

```
class MyClass
(
[RegistryKey (HKEYCURRENTUSER, "foo")]
 public int Foo;
)
```

Чтобы прикрепить определенный атрибут к типу или члену C#, нужно просто задать данные атрибута в скобках перед целевым типом или членом. В нашем примере мы прикрепили атрибут RegistryKey к полю MyClass.Foo. Как вы вскоре увидите, все, что нам надо сделать в период выполнения,— это запросить значение поля, связанное с разделом реестра, и использовать его, чтобы сохранить данные в реестре.

ОПРЕДЕЛЕНИЕ АТРИБУТОВ

В предыдущем примере синтаксис прикрепления атрибута к типу или члену похож на тот, что применяется при создании экземпляра класса. Дело в том, что атрибут на самом деле является классом, производным от базового класса System.Attribute.

А теперь немного расширим атрибут RegistryKey:

```
public enum RegistryHives
{
  HKEY_CLASSES_ROOT,
  HKEY_CURRENT_USER,
  HKEY_LOCAL_MACHINE,
  HKEY_CURRENT_CONFIG
}
```

```
public class RegistryKeyAttribute : Attribute
{
  public RegistryKeyAttribute ( RegistryHives  Hive, String ValueName)
  {
    this.Hive = Hive;
    this.ValueName = ValueName;
  )
  protected RegistryHives hive;
  public RegistryHives Hive
  {
    return hive;
  }
  set
  {
    hive = value;
  }
  protected String valueName;
  public String ValueName
  {
    get
    {
      return valueName e;
    }
    set
    {
      valueName - value;
    }
  }
}
```

В этом примере были добавлены enum для различных типов разделов реестра, конструктор для класса-атрибута (который принимает тип и имя раздела реестра), а также два свойства для имени улья реестра и имени значения. Теперь мы видим более широкие возможности атрибутов, так что посмотрим, как запросить атрибут в период выполнения. Мы будем работать с полностью функциональным примером. Перейдем к рассмотрению некоторых более сложных вопросов, связанных с определением и прикреплением атрибутов.

В приведенных примерах к именам классов атрибутов добавлено слово Attribute. Однако, прикрепляя атрибут к типу или члену, можно отбросить этот суффикс. Это одна из возможностей сокращения, изысканная разработчиками C#, и дается она нам даром. Обнаружив атрибут, прикрепленный к типу или члену, компилятор будет искать класс с именем заданного атрибута, производный от System.Attribute. Если класс найти не удастся, компилятор добавит к имени заданного атрибута слово Attribute и будет продолжать поиск получившегося имени. Поэтому в повседневной практике имена классов атрибутов при определении оканчиваются словом Attribute. Впоследствии эта часть имени опускается.

ЗАПРОС ИНФОРМАЦИИ ОБ АТРИБУТАХ

Мы знаем, как определять атрибуты в виде производных от System.Attribute и как прикреплять их к типу или члену. А что теперь? Как использовать атрибуты при программировании? Короче, как производится запрос типа или члена (как прикрепленных к ним атрибутов, так и их параметров)?

Чтобы запросить тип или член о прикрепленных к ним атрибутах, нужно применить отражение (reflection).

Отражение — это набор функций, позволяющих в период выполнения динамически определять характеристики типов в приложении. Например, с помощью Reflection API из состава .NET Framework можно циклически запрашивать метаданные всего приложения и создавать списки определенных для него классов, типов и методов. Рассмотрим несколько примеров атрибутов и способов их запроса с помощью отражения.

Атрибуты класса

Способ получения атрибута зависит от типа члена, к которому производится запрос. Допустим, вам нужно узнать атрибут, определяющий удаленный сервер, на котором должен быть создан объект. Без атрибутов вам бы пришлось сохранять эту информацию в константе или файле ресурсов приложения. Используя же их, можно просто создать аннотацию для класса с именем удаленного сервера, например, так:

```
using System;
public enum RemoteServers
{
  DC,
  BIND,
  COCKER
}
public class RemoteObjectAttribute: Attribute
{
  public RemoteObjectAttribute(RemoteServers Server)
  {
    this.server = Server;
  }
  protected RemoteServers server;
  public string Server
  {
    get
    {
      return RemoteServers.GetName(typeof(RemoteServers), this.server!;
    }
  }
}
```

```
[RemoteObject(RemoteServers.COOKER)]
class MyRemotableClass
{
}
```

Сервер, на котором необходимо создавать объект, можно определить так:

```
class ClassAttrApp
{
  public static void Main()
  {
    Type type = typeof(MyRemotableClass);
    foreach (Attribute attr in type.GetCustomAttributes(true))
    {
      RemoteObjectAttribute remoteAttr = attr as RemoteObjectAttribute;
      if (null != remoteAttr)
      {
        Console.WriteLine("Создайте этот объект на {0}.", remoteAttr. Server);
      }
    }
  }
}
```

Как можно ожидать, приложение выдаст следующее:

Создайте этот объект на COOKER.

Поскольку все вариации данного примера будут использовать общий код, рассмотрим, что здесь происходит с точки зрения отражения и как оно возвращает значение атрибута в период выполнения.

Первая строка в методе Main использует оператор typeof:

```
Type type = typeof (MyRemotableClass);
```

Этот оператор возвращает ассоциированный с типом объект System.Type, который передается как его единственный аргумент. Как только этот объект оказался в вашем распоряжении, к нему можно выполнить запрос.

Относительно следующей строки в пояснении нуждаются два момента:

```
foreach (Attribute attr in type.GetCustomAttributes (true) )
```

Первый — вызов метода Type.GetCustomAttributes. Этот метод возвращает массив значений типа Attribute, который в данном случае будет содержать все атрибуты, прикрепленные к классу MyRemotableClass. Второй — оператор foreach, циклически обрабатывающий возвращенный массив, помещая каждое последовательное значение в переменную attr типа Attribute.

Следующее выражение использует оператор as, чтобы попытаться преобразовать переменную attr в тип RemoteObjectAttribute:

```
RemoteObjectAttribute remoteAttr = attr as RemoteObjectAttribute;
```

Далее выполняется проверка на пустое значение, которое указывает на наличие сбоя при использовании оператора as. Если значение не пустое, значит, переменная remoteAttr содержит верный атрибут, прикрепленный к типу MyRemotableClass — мы вызываем одно из свойств RemoteObjectAttribute, чтобы вывести имя удаленного сервера:

```
if (null != remoteAttr)
```

```
{
Console.WriteLine("Создайте этот объект на {0}", remoteAttr.Server);
}
```

Атрибуты поля

В качестве последнего примера запроса членов как прикрепленных к ним атрибутов мы рассмотрим способ запроса полей класса. Допустим, наш класс содержит поля, значения которых нужно сохранить в реестре. Для этого можно определить атрибут с конструктором, принимающим как параметр enum с ульями реестра, и строку, представляющую имя параметра реестра. Затем можно выполнить запрос к полю как к разделу реестра:

```
using System;
using System.Reflection;
public enum RegistryHives
{
HKEYCLASSES_ROOT,
HKEYCURRENT_USER,
HKEYLOCAL_MACHINE,
HKEY_USERS,
HKEYCURRENT_CONFIG
)
public class RegistryKeyAttribute: Attribute
{
  public RegistryKeyAttribute (RegistryHives Hive, String ValueName)
  {
    this.Hive = Hive;
  } this.ValueName = ValueName;
  protected RegistryHives hive;
  public RegistryHives Hive
  <
    get { return hive; }
    set ( hive = value; )
  }
  protected String valueName;
  public String ValueName
  {
    get { return valueName; }
    set { valueName = value; )
  }
}
class Tes=Class
{
  [RegistryKey(RegistryHives.HKEYCURRENT_USER, "Foo")]
  public int Foo;
  public int Bar;
}
```

```
class FieldAttrApp
{
  public static void Main()
  {
    Type type = Type.GetType("TestClass");
    foreach(FieldInfo field in type.GetFields())
    {
      foreach (Attribute attr in field.GetCustomAttributes(true))
      {
        RegistryKeyAttribute registryKeyAttr = attr as RegistryKeyAttribute;
        if(null != registryKeyAttr)
        {
          Console.WriteLine("{0} будет сохранен в {1}\\\\{2}",field.Name,
        }     registryKeyAttr.Hive, registryKeyAttr.ValueName);
      }
    }
  }
}
```

Результат работы программы:

Foo будет сохранен в HKEYCURRENT_USER\\Foo

Этот пример в чем-то дублирует предыдущий. Однако пара деталей все же важна для нас. Как и объект MethodInfo, определенный для получения информации о методе из объекта типа, объект FieldInfo предоставляет аналогичную функциональность для получения из объекта сведений о поле. Как и в предыдущем примере, мы начнем с получения объекта типа, ассоциированного с нашим тестовым классом. Затем мы циклически обработаем массив FieldInfo, а также все атрибуты каждого объекта FieldInfo, пока не найдем нужный — RegistryKeyAttribute. Если мы его обнаружим, то выведем имя поля и значения полей атрибута Hive и ValueName.

ПАРАМЕТРЫ АТРИБУТОВ

В вышеприведенном примере было рассказано об использовании прикрепленных атрибутов с помощью их конструкторов. А теперь рассмотрим некоторые вопросы, связанные с конструкторами атрибутов, о которых раньше не упоминалось.

Типы параметров

В примере FieldAttrApp вы могли видеть атрибут с именем RegistryKeyAttribute. Его конструктор имел такой вид:

```
public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
```

Далее к полю был прикреплен атрибут на основе сигнатуры этого конструктора:

```
[RegistryKey(RegistryHives.HKEYCURENT_USER,"Foo")]
public int Foo;
```

Пока все просто. У конструктора есть два параметра, и два параметра использовались, чтобы прикрепить этот атрибут к полю. Однако мы можем упростить код. Если параметр большую часть времени останется неизменным, зачем каждый раз заставлять пользователя класса типизировать его? Мы можем установить значения по умолчанию, применив позиционные и именованные параметры.

Позиционными называются параметры конструктора атрибута. Они обязательны и должны задаваться каждый раз при использовании атрибута. В нашем примере RegistryKeyAttribute позиционными являются оба параметра, Hive и ValueName. Именованные параметры на самом деле не определяются в конструкторе атрибута. Они скорее представляют собой нестатические поля и свойства. Поэтому именованные параметры позволяют клиенту устанавливать значения полей и свойств атрибута при создании его экземпляра, не требуя от вас создания конструктора для каждой возможной комбинации полей и свойств, значения которых может понадобиться установить клиенту.

Каждый открытый конструктор может определять последовательность позиционных параметров. Это верно и в отношении любого типа класса. Но в случае атрибутов после указания позиционных параметров пользователь может ссылаться на некоторые поля или свойства, применяя синтаксис

```
Имя_поля_или_свойства=Значение.
```

Чтобы проиллюстрировать это, изменим атрибут RegistryKeyAttribute. Мы создадим лишь один позиционный параметр RegistryKeyAttribute.ValueName, а необязательным именованным параметром будет RegistryKeyAttribute.Hive. Итак, возникает вопрос: «Как определить что-либо как именованный параметр?». Поскольку в определение конструктора включены только позиционные и поэтому необходимые параметры, просто удалите параметр из определения конструктора. Впоследствии пользователь может указывать как именованный параметр любое поле, не являющееся readonly, static или const, или любое поле, у которого есть метод-аксессор для установки его значения, или установщик, который не является статическим. Поэтому, чтобы сделать RegistryKeyAttribute.Hive именованным параметром, мы уберем его из определения конструктора, так как он уже существует в виде открытого свойства, доступного для чтения и записи:

```
public RegistryKeyAttribute(String ValueName)
```

Теперь пользователь может прикрепить атрибут любым из следующих способов:

```
[RegistryKey("Foo")]
[RegistryKey("Foo", Hive = RegistryHives.HKEYLOCAL_MACHINE)]
```

Это дает гибкость, обеспечиваемую наличием у поля значения по умолчанию, в то же время предоставляя пользователю возможность изменять это значение при необходимости. Но если пользователь не устанавливает значения поля RegistryKeyAttribute.Hive, как мы зафиксируем для него значение по умолчанию? Вы можете подумать: «Хорошо, посмот-

рим, не установлено ли оно в конструкторе». Однако проблема в том, что RegistryKeyAttribute.Hive — это enum, в основе которого лежит int — размерный тип. Это значит, что по умолчанию компилятор инициализирует его значением 0! Если мы изучим значение RegistryKeyAttribute.Hive в конструкторе и найдем его равным 0, то не сможем узнать, установлено ли оно вызывающим кодом через именованный параметр или инициализировано компилятором как размерный тип. К сожалению, единственный способ решения проблемы — это изменить код так, чтобы значение, равное 0, стало неверным. Это можно сделать, модифицировав RegistryHives enum:

```
public enum RegistryHives
 {
HKEYCLASSES_ROOT =1,
HKEYCURRENT_USER, HKEYLOCAL_MACHINE,
HKEY_USERS,
HKEYCURRENT_CONFIG
}
```

Теперь мы знаем, что единственный способ, позволяющий RegistryKeyAttribute.Hive быть равным 0,— инициализация его компилятором этим значением, если после пользователь не изменил его значение через именованный параметр. Сейчас мы можем написать для инициализации примерно такой код:

```
public RegistryKeyAttribute( String ValueName)
{
if (this.Hive == 0 )
 this. Hive = RegistryHives .HKEYCURRENT_USER;
this.ValueName = ValueName;
]
```

Используя именованные параметры, вы должны указать сначала позиционные параметры. После этого именованные параметры можно перечислить в любом порядке, так как перед ними идет имя поля или свойства. При компиляции приведенного ниже примера возникнет ошибка компилятора:

```
// Это ошибочный код, поскольку позиционные параметры не могут
II стоять после именованных параметров.
[RegistryKey (Hive=RegistryHives .HKEYLOCAL_MACHIHE, "Foo") ]
```

Кроме того, вы не можете именовать позиционные параметры. При компиляции атрибутов компилятор сначала попытается разрешить именованные параметры, затем разрешить оставшиеся именованные параметры с помощью сигнатуры метода. Хотя компилятор сможет разрешить каждый именованный параметр, следующий код не будет компилироваться, так как по завершении разрешения именованных параметров компилятор не найдет ни одного позиционного параметра и выдаст сообщение «No overload for method 'RegistryKeyAttribute' takes «0» arguments»:

```
[RegistryKey(ValueName="Foo", Hive = RegistryHives.HKEYLOCAL_MACHINE) ]
```

Наконец, именованными параметрами могут быть любые открытые поля или свойства, включая метод-установщик, не определенные как static или const.

Типы позиционных и именованных параметров класса атрибута ограничены следующим набором:

- bool, byte, char, double, float, int, long, short, string;
- System.Type;
- object;
- enum — при условии, что он и все типы, по отношению к которым он является вложенным, открытые (как в примере, где используется перечисление ульев реестра);
- одномерный массив значений любого из вышеуказанных типов.

Поскольку набор типов параметров ограничен приведенным выше списком, вы не можете передавать конструктору структуры данных наподобие класса. Это ограничение имеет смысл, так как атрибуты прикрепляются в период разработки и в это время у вас не будет созданного экземпляра класса (объекта). Допускается применение вышеуказанных типов, так как они позволяют жестко запрограммировать их значения во время разработки.

ТИПЫ АТРИБУТОВ

Кроме пользовательских параметров, которые вы задаете для аннотации обычных типов C#, с помощью атрибута AttributeUsage можно определить способ применения этих атрибутов. Согласно документации правила вызова атрибута AttributeUsage таковы;

```
[AttributeUsage (
validon,
AllowMultiple = allowmultiple,
Inherited = Inherited
)]
```

Как видите, позиционные параметры легко отличить от именованных. Настоятельно рекомендуется так документировать ваши атрибуты, чтобы у их пользователя не возникало необходимости обращаться к исходному коду класса атрибута для поиска открытых полей, доступных для чтения и записи, которые могут применяться как именованные атрибуты.

Определение целевого типа атрибута

А сейчас снова взгляните на атрибут AttributeUsage из предыдущего раздела. Заметьте: параметр validon является позиционным и, естественно, обязательным. Он позволяет задавать типы, к которым может быть прикреплен атрибут. На самом деле тип параметра validon атрибута

AttributeUsage — AttributeTargets, представляющий собой перечисление, определяемое так:

```
public enum AttributeTargets
{
Assembly = 0x0001,
Module = 0x0002,
Class = 0x0004,
Struct = 0x0008,
Enum = 0x0010,
Constructor = 0x0020,
Method = 0x0040,
Property = 0x0080,
Field = 0x0100,
Event = 0x0200,
Interface = 0x0400,
Parameter = 0x0800,
Delegate = 0x1000,
All = Assembly [ Module | Class | Struct | Enun | Constructor |
Method | Property | Field | Event | Interface | Parameter |
Delegate,
ClassMembers = Class | Struct | Enum | Constructor | Method ! Property | Field
1 Event | Delegate | Interface;
}
```

При использовании атрибута AttributeUsage можно заявить: AttributeTargets.All. Это позволяет прикрепить атрибут к любому из типов в списке перечисления AttributeTargets. Если вы вообще не определили AttributeUsage, можете прикрепить атрибут к любому типу — это значение по умолчанию. И тут вы можете спросить: «А зачем вообще значение validon?» А затем, что с вашим атрибутом могут применяться именованные параметры, которые вы, может быть, захотите изменить. Помните: используя именованный параметр, вы должны поставить все позиционные параметры перед ним. Это позволяет легко задавать применение атрибутов по умолчанию, определяемое AttributeTargets.AII, и при этом устанавливать именованные параметры.

Итак, когда и для чего задавать параметр validon (AtributeTargets)? Когда вам нужно точно контролировать способы применения атрибута. В наших примерах мы создали атрибут RemoteObjectAttribute, применимый только к классам, атрибут TransactionableAttribute, применимый только к методам, и атрибут RegistryKeyAttribute, который имеет смысл использовать лишь по отношению к полям. Если мы хотим убедиться, что они были использованы для аннотации только тех типов, для которых разработаны, то можем определить их так (для ясности тела атрибутов не приводятся):

```
[AttributeUsage(AttributeTargets.Class)]
[public class RemoteObjectAttribute: Attribute]
[ … }
```

```
[AttributeUsage (AttributeTargets.Method)]
[public class TransactionableAttribute: Attribute]
{ … }

[AttributeUsage (AttributeTargets.Field) ]
[public class RegistryKeyAttribute: Attribute]
{ … )
```

И последний момент относительно перечисления AttributeTargets. Вы можете комбинировать члены с помощью оператора !. Если у вас есть атрибут, применимый и к полям, и к свойствам, AttributeUsage можно прикрепить так:

```
[AttributeUsage (AttributeTargets.Field | AttributeTargets.Property) ]
```

Атрибуты однократного и многократного использования

AttributeUsage позволяет задать атрибуты для одно- и многократного применения. Это определит, как много раз один атрибут может быть задействован с одним полем. По умолчанию все атрибуты используются однократно. Это означает, что при компиляции следующего кода возникнет ошибка компилятора:

```
using System;
using System.Reflection;
public class SingleUseAttribute: Attribute
{
  public SingleUseAttrlbute (String str)
    // ОШИБКА: возникает ошибка компилятора "duplicate attribute"
    [SingleUse("abc")]
      [SingleUse("def")]
      class MyClass
    {
    }
  class SingleUseApp
  i
    public static void Main ()
    {
    }
  }
}
```

Чтобы исправить эту ошибку, укажите в строке AttributeUsage, что хотите разрешить многократное использование атрибута с данным типом. Такой код будет работать:

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
public class SingleUseAttribute: Attribute
```

```
{
  public SingleUseAttribute(String str)
  {
  }
}

[SingleUse("abc")]
[SingleUse("def")]
class MyClass
{
}

class SingleUseApp
{
  public static void Main f)
  {
  }
}
```

Практическим примером использования этого подхода служит атрибут RegistryKeyAttribute (см. подраздел «Определение атрибутов»). Поскольку вполне реально, что поле может быть сохранено в нескольких местах реестра, попробуйте прикрепить атрибут Attribute Usage с именованным параметром AllowMultiple, как показано в примере выше.

Наследование атрибутов

Последний параметр атрибута AttributeUsage — флаг inherited, определяет, может ли атрибут наследоваться. Его значение по умолчанию равно false. Если же установить флаг inherited в true, его значение зависит от значения флага AllowMultiple. Если inherited находится в true, а AllowMultiple — в false, установленный атрибут заменит унаследованный. Однако если и inherited, и AllowMultiple установлены в true, атрибуты члена будут аккумулироваться, то есть собираться вместе.

ИДЕНТИФИКАТОРЫ АТРИБУТОВ

Взгляните на следующий код и попробуйте определить, что аннотирует атрибут — возвращаемое значение или метод:

```
class MyClass
{
[HRESULT]
public long Foo();
1
```

Если у вас есть опыт работы с COM, вы должны знать, что HRESULT — это стандартный возвращаемый тип для всех методов, кроме AddRef или

Release. Однако нетрудно заметить, что если имя атрибута применимо и к возвращаемому значению, и к имени метода, то компилятору будут непонятны ваши намерения. Вот несколько сценариев, в которых компилятор не поймет ваших намерений из контекста:

- метод или возвращаемый тип;
- событие, поле или свойство;
- делегат или возвращаемый тип;
- свойство, аксессор, возвращаемое значение метода-получателя или параметр значения установщика.

В каждом из этих случаев компилятор производит определение на основе того, что считается «наиболее употребительным». Чтобы обойти такой путь принятия решения, используйте идентификаторы атрибута: assembly, module, type, method, property, event, field, param, return.

Чтобы воспользоваться идентификатором атрибута, поставьте перед именем атрибута идентификатор и двоеточие. В примере MyClass, чтобы быть уверенным в том, что компилятор сможет определить HRESULT как атрибут, аннотирующий возвращаемое значение, а не как метод, вы должны задать его следующим образом:

```
class MyClass
{
[return: HRESULT]
public long Foo( );
1
```

15. ИНТЕРФЕЙСЫ

Ключ к пониманию интерфейсов лежит в их сравнении с классами. Классы — это объекты, обладающие свойствами и методами, которые на эти свойства воздействуют. Хотя классы проявляют некоторые характеристики, связанные с поведением (методы), они представляют собой предметы, а не действия, присущие интерфейсам. Интерфейсы же позволяют определять характеристики или возможности действий и применять их к классам независимо от иерархии последних. Допустим, у вас есть дистрибьюторское приложение, составляющие которого можно упорядочить. Среди них могут быть классы Customer, Supplier и Invoice. Некоторые другие, скажем, MaintenanceView или Document, упорядочивать не надо. Как упорядочить только выбранные вами классы? Очевидный способ — создать базовый класс с именем типа Serializable. Но у этого подхода есть большой минус: одна ветвь наследования здесь не подходит, так как нам не требуется наследование всех особенностей поведения. C# не поддерживает множественное наследование, так что невозможно произвести данный класс от нескольких классов. А вот интерфейсы позволяют определять набор семантически связанных методов и свойств, способных реализовать избранные классы независимо от их иерархии.

Концептуально интерфейсы представляют собой связки между двумя в корне отличными частями кода. Иначе говоря, при наличии интерфейса и класса, определенного как реализующий данный интерфейс, клиентам класса дается гарантия, что у класса реализованы все методы, определенные в интерфейсе. Скоро вы это поймете на примерах.

Прочитав данную главу, вы поймете, почему интерфейсы являются такой важной частью C# в частности и программирования на основе компонентов вообще. Затем мы познакомимся с объявлением и реализацией интерфейсов в приложениях на C#. В завершение мы углубимся в специфику использования интерфейсов и преодоления врожденных проблем с множественным наследованием и конфликтами имен.

Когда вы создаете интерфейс и в определении класса задаете его использование, говорят, что класс реализует интерфейс. Интерфейсы — это набор характеристик поведения, а класс определяется как реализующий их.

ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСОВ

Чтобы понять, где интерфейсы приносят пользу, рассмотрим традиционную проблему программирования в Windows, когда нужно обеспечить универсальное взаимодействие двух совершенно различных фрагментов

кода без использования интерфейсов. Представьте себе, что вы работаете на Microsoft и являетесь ведущим программистом команды по разработке панели управления. Вам надо предоставить универсальные средства, которые дают возможность клиентским апплетам «закрепляться» на панели управления, показывая при этом свой значок и позволяя клиентам выполнять их. Если учесть, что эта функциональность разрабатывалась до появления COM, возникает вопрос: как создать средства интеграции любых будущих приложений с панелью управления? Задуманное решение долгие годы было стандартной частью разработки Windows.

Как ведущий программист по разработке панели управления, вы создаете и документируете функцию (функции), которая должна быть реализована в клиентском приложении, и некоторые правила. В случае апплетов панели управления, Microsoft определила, что для их написания вам нужно создать динамически подключаемую библиотеку, которая реализует и экспортирует функцию CPLApplet. Вам также потребуется добавить к имени этой DLL расширение .cpl и поместить ее в папку Windows System32 (для Windows ME или Windows 98 это будет Windows\System32, а для Windows 2000 — WINNT\System32). При загрузке панель управления загружает все DLL с расширением .cpl из папки System32 (с помощью функции LoadLibrary), а затем вызывает функцию GetProcAddress для загрузки функции CPLApplet, проверяя таким образом выполнение вами соответствующих правил и возможность корректного взаимодействия с панелью управления.

Как уже говорилось, эта стандартная модель программирования в Windows позволяет выйти из ситуации, когда вы хотите, чтобы ваш код универсальным образом взаимодействовал с кодом, который будет разработан в будущем. Однако это не самое элегантное решение. Главный недостаток этой методики в том, что она вынуждает включать в состав клиента — в данном случае в код панели управления — большие порции проверяющего кода. Например, панель управления не может просто полагаться на допущение, что каждый .cpl-файл в папке является DLL Windows. Панель управления также должна проверить наличие в этой DLL функций коррекции и что эти функции делают именно то, что описано в документации. Здесь-то интерфейсы и вступают в дело. Интерфейсы позволяют создавать такие же средства, связывающие несовместимые фрагменты кода, но при этом они более объектно-ориентированны и гибки. Кроме того, поскольку интерфейсы являются частью языка C#, компилятор гарантирует, что если класс определен как реализующий данный интерфейс, то он выполняет именно те действия, о которых заявляет, что должен их выполнять.

В C# интерфейс — понятие первостепенной важности, объявляющее ссылочный тип, который включает только объявления методов. Но что значит «понятие первостепенной важности»? Необходимо сказать, что эта встроенная функция является неотъемлемой частью языка. Иначе говоря, это не то, что было добавлено позже, после того как разработка языка была закончена. Давайте подробнее познакомимся с интерфейсами, узнаем, что они собой представляют и как их объявлять.

ОБЪЯВЛЕНИЕ ИНТЕРФЕЙСОВ

Интерфейсы могут содержать методы, свойства, индексаторы и события, но ни одна из этих сущностей не реализуется в самом интерфейсе. Рассмотрим их применение на примере. Допустим, вы создаете для вашей компании редактор, содержащий элементы управления Windows. Вы пишете редактор и тестовые программы для проверки элементов управления, размещаемых пользователями на форме редактора. Остальная часть команды пишет элементы управления, которыми будет заполнена форма. Вам почти наверняка понадобятся средства проверки на уровне формы. В определенное время, скажем, когда пользователь явно приказывает форме проверить все элементы управления или при обработке формы, последняя циклически проверяет все прикрепленные к ней элементы управления, или, что более подходяще, заставляет элемент управления проверить самого себя.

Как предоставить такую возможность проверки элемента управления? Именно здесь проявляется превосходство интерфейсов. Вот пример простого интерфейса с единственным методом Validate:

```
interface IValidate
{
bool Validate();
}
```

Теперь вы можете задокументировать тот факт, что если элемент управления реализует интерфейс IValidate, то этот элемент управления может быть проверен.

Рассмотрим пару аспектов, связанных с приведенным кодом. Вы не должны задавать для метода интерфейса модификатор доступа, такой как public. При указании модификатора доступа перед объявлением метода возникает ошибка периода компиляции. Дело в том, что все методы интерфейса открыты по умолчанию (разработчики на C++ могут также заметить, что, поскольку интерфейсы по определению — это абстрактные классы, не требуется явно объявлять метод как чисто виртуальный (pure virtual), прибавляя «= 0» к его определению).

Кроме методов, интерфейсы могут определять свойства, индексаторы и события:

```
interface IExampleInterface
{
// Пример объявления свойства. .
int testProperty { get; }
// Пример объявления события.
event testEvent Changed;
// Пример объявления индексатора
 string this[int index] { get; set; }
}
```

СОЗДАНИЕ ИНТЕРФЕЙСОВ

Поскольку интерфейс определяет связь между фрагментами кода, любой класс, реализующий интерфейс, должен определять любой и каждый элемент этого интерфейса, иначе код не будет компилироваться. Используя IValidate из нашего примера, клиентский класс должен реализовать лишь методы интерфейса. В следующем примере есть базовый класс FancyControl и интерфейс IValidate. Кроме того, в нем имеется класс MyControi, производный от FancyControl, реализующий интерфейс IValidate. Обратите внимание на синтаксис и способ приведения объекта MyControi к интерфейсу IValidate для ссылки на его члены.

```
using System;

public class FancyControl
{
  protected string Data;
  public string data
  {
    get
    {
      return this.Data;
    }
    set
    {
      this.Data = value;
    }
  }
}

interface IValidate
{
  bool Validate();
}

class MyControl: FancyControl, IValidate
{
  public MyControl()
  {
    data = "таблица данных";
  }
  public bool Validate()
  {
    Console.WriteLine("Проверка...{0}", data);
    return true;
  }
}
```

```
class InterfaceApp
{
  public static void Main()
  {
    MyControl myControl = new MyControl();

    IValidate val = (IValidate)myControl;
    bool success = val.Validate();
    Console.WriteLine("Проверка '{0}' {1} была завершена успешно",
      myControl.data,
      (true == success ? "": "не"));
  }
}
```

С помощью определения такого класса и интерфейса редактор может запрашивать у элемента управления, реализует ли он интерфейс IValidate (ниже будет показано, как это сделать). Если это так, редактор может проверить данный элемент управления, а затем вызывать реализованные методы интерфейса. Вы можете спросить: «А почему бы мне просто не определить базовый класс для использования в этом редакторе, у которого есть чисто виртуальная функция Validate? Ведь после этого редактор будет принимать только элементы управления, производные от этого базового класса, да?»

Да, но... это решение повлечет суровые ограничения. Допустим, вы создаете собственные элементы управления, каждый из которых является производным от гипотетического базового класса. Как следствие, все они реализуют виртуальный метод Validate. Это будет работать, пока в один прекрасный день вы не найдете по-настоящему замечательный элемент управления и вам не захочется использовать его в редакторе. Допустим, вы нашли компонент «сетка», написанный кем-то другим и поэтому не являющийся производным от нужного редактору базового класса «элемент управления». На C++ решение в том, чтобы с помощью множественного наследования сделать сетку производной от компонента стороннего разработчика и базового класса редактора одновременно. Но C# не поддерживает множественное наследование.

Интерфейсы позволяют реализовать несколько характеристик поведения в одном классе. На C# можно создавать объекты, производные от одного класса, и в дополнение к этой унаследованной функциональности реализовать столько интерфейсов, сколько нужно для класса. Например, чтобы приложение-редактор, проверяя содержимое элемента управления, связывало элемент управления с базой данных и последовательно направляло его содержимое на диск, объявите свой класс так:

```
public class MyGrid: ThirdPartyGrid, IValidate, ISeriallzable, IDataBound
```

И все же вопрос остался: «Как отдельный фрагмент кода узнает, реализован ли классом данный интерфейс?»

Инструкция is

В предыдущем примере вы видели код, использованный для приведения объекта (MyControl) к одному из реализованных в нем интерфейсов (IValidate) и затем для вызова одного из членов этого интерфейса (Validate).

```
MyControl myControl = new MyControl ();
IValidate val= (IValidate) myControl;
bool Success = val.Vailidate();
```

Что будет, если клиент попытается использовать класс так, как если бы в последнем был реализован метод, на самом деле в нем не реализованный? Следующий пример будет скомпилирован, поскольку интерфейс ISerializable является допустимым. И все же в период выполнения будет передано исключение System.InvalidCastException, так как в MyGrid не реализован интерфейс ISerializable. После этого выполнение приложения прервется, если только исключение не будет явно уловлено.

```
using System;

public class FancyControl
{
  protected string Data;
  public string data
  {
    get
    t
      return this.Data;
    }
    set
    {
      this.Data = value;
    }
  }
}

interface ISerializable
i
  bool Save();
}
interface IValidate
{
  bool Validate();
}

class MyControl: FancyControl, IValidate
{
  public MyControl()
```

```
  {
    data = "таблица данных";
  }

  public bool Validated
  {
    Console.WriteLine("Проверка... {0}", data);
    return true;
  }
}

class IsOperator1App
{
  public static void Main()
  {
    MyControl myControl = new MyControl();
    ISerializable ser = (ISerializable)myControl;

    //будет сгенерировано исключение
    bool success = ser.Save();

    Console.WriteLine("Сохранение '{0}' {1} завершено успешно",
      myControl.data,
      (true == success ? "": "не"));
  }
}
```

Конечно, улавливание исключения не повлияет на то, что предназначенный для выполнения код в этом случае не будет исполнен. Способ запроса объекта перед попыткой его приведения — вот что вам нужно. Один из способов — задействовать оператор is. Он позволяет в период выполнения проверять совместимость одного типа с другим. Оператор имеет следующий вид, где «выражение» — ссылочный тип:

```
выражение is тип
```

Результат оператора is — булево значение, которое затем можно использовать с условными операторами. В следующем примере код был изменен, чтобы проверять совместимость между классом MyControl и интерфейсом ISerializable перед попыткой вызова метода ISerializable:

```
using System;

public class FancyControl
{
  protected string Data;
  public string data
  {
```

```
    get
    (
      return this.Data;
    }
    set
    {
      this.Data = value;
    }
  }
}

interface ISerializable
{
  bool Save();
}

interface IValidate
{
  bool Validate();
}

class MyControl: FancyControl, IValidate
{
  public MyControl()
  {
    data = "таблица данных";
  )

  public bool Validated
  {
    Console.WriteLine("Проверка... {0}", data);
    return true;
  )
)

class IsOperator1App
{
  public static void Main()
  {
    MyControl myControl = new MyControl();

    if(myControl is ISerializable)

      ISerializable ser = (ISerializable)myControl;

      //будет сгенерировано исключение
```

```
      bool success = ser.Save();

      Console.WriteLine("Сохранение ' (0}' {1} завершено успешно",
        myControl.data,
        (true == success ? "": "не")];
    }
    else
    {
      Console.WriteLine("Интерфейс ISerializable не реплизован");
    }
  }
}
```

После запуска приложения вы получите сообщение:

Интерфейс ISerializable не реализован

Вы увидели, как оператор is позволяет проверить совместимость двух типов, чтобы гарантировать их корректное использование. А теперь рассмотрим его близкого родственника — оператор as и сравним их.

Инструкция as

Код операции isinst, генерируемый компилятором для оператора C# is, проверяет, чем является объект: экземпляром класса или интерфейсом. И лишь после этого, при условии, что проверка условий пройдена, компилятор генерирует код операции приведения типа и выполняет приведение объекта к типу интерфейса. Операция приведения типа осуществляет собственную проверку, и, поскольку она работает несколько иначе, в результате сгенерированный код выполняет неэффективную работу, дважды проверяя правильность приведения.

Мы можем повысить эффективность процесса проверки с помощью оператора as, который преобразует совместимые типы и принимает такой вид:

```
объект = выражение as тип,
```

где «выражение» — это любой ссылочный тип.

Можно думать, что оператор as представляет собой комбинацию оператора is и, если рассматриваемые типы совместимы, приведения. Важное различие между as и is в том, что если выражение и тип несовместимы, то вместо возврата булева значения оператор as устанавливает объект в null. Теперь наш пример можно переписать:

```
using System;

public class FancyControl
{
  protected string Data;
  public string data
  f
    get
```

```
    {
      return this.Data;
    }
    set
    {
      this.Data = value;
    }
  }
}

interface ISerializable
{
  bool Save();
)

interface IValidate
{
  bool Validate() ;
}

class MyControl: FancyControl, IValidate
{
  public MyControl()
  {
    data = "таблица данных";
  }

  public bool Validate()
  {
    Console.WriteLine("Проверка... {0}", data);
    return true;
  )
)

class AsOperatorlApp
{
  public static void Main()
  f
    KyControl myControl = new MyControl();

    ISerializable ser = myControl as ISerializable;
    if (ser != null)
    {
      //будет сгенерировано исключение
      bool success = ser.Save();

      Console.WriteLine("Сохранение '{0}' {1} завершено успешно",
```

```
        myControl.data,
        (true == success ? "": "не"));
    }
    else
    <
      Console.WriteLine("Интерфейс ISerializable не реализован");
    }
  }
}
```

Проверка, гарантирующая правильность приведения, производится только раз, что, естественно, намного эффективнее.

ЯВНАЯ КВАЛИФИКАЦИЯ ИМЕНИ ЧЛЕНА ИНТЕРФЕЙСА

Вы уже познакомились с классами, реализующими интерфейсы путем задания модификатора доступа public, за которым следует сигнатура метода интерфейса. Однако иногда у вас будет возникать желание (или даже необходимость) явно квалифицировать имя члена именем интерфейса. В этом разделе мы изучим две причины, которые могут заставить вас поступить таким образом.

Сокрытие имен с помощью интерфейсов

Чтобы вызвать метод, реализованный в интерфейсе, необходимо привести экземпляр этого класса к типу интерфейса и вызвать нужный метод — это самый распространенный подход. Хотя это работает и многие используют данную методику, формально вы вовсе не обязаны приводить объект к реализованному им интерфейсу, чтобы вызывать методы этого интерфейса. Это так, потому что методы интерфейса, реализованные классом, также являются открытыми методами класса. Взгляните на код на C#, особенно на метод Main, чтобы понять, что я имею в виду:

```
using System;

public interface IDataBound
{
  void Bind();
}

public class EditBox: IDataBound
{
  //реализация IDataBound
  public void Bind()
  {
    Console.WriteLine("Связывание с данными... " );
  \
1
```

```
class NameHidding1App
{
  public static void Main()
  {
    EditBox edit = new EditBox();
    Console.WriteLine("Вызов метода EditBox.Bind()");
    edit.Bind();

    Console.WriteLine();

    IDataBound bound = (IDataBound)edit;
    Console.WriteLine("Вызов метода (IDataBound)EditBox.Bind()");
    bound.Bind();
  }
}
```

Теперь этот пример выдаст следующее:

Вызов метода EditBox.Bind()
Связывание с данными...
Вызов метода (IDataBound)EditBox.Bind()
Связывание с данными...

Заметьте, хотя это приложение вызывает реализованный метод Bind двумя способами — с помощью приведения и без него, оба вызова корректно функционируют — Bind выполняется. Хотя поначалу возможность прямого вызова реализованного метода без приведения объекта к интерфейсу может показаться неплохой, порой это более чем нежелательно. Самая очевидная причина в том, что реализация нескольких интерфейсов, каждый из которых может содержать массу членов, может привести к быстрому засорению открытого пространства имен вашего класса членами, не имеющими значения за пределами видимости реализующего эти интерфейсы класса. Вы можете не позволять реализованным членам интерфейсов становиться открытыми членами класса, применяя методику сокрытия имен (name hiding).

В простейшем случае сокрытие имен представляет собой возможность скрывать имя унаследованного члена от любого кода за пределами производного или реализующего его класса (его часто называют внешним миром (outside world). Возьмем тот пример, где класс EditBox должен был реализовывать интерфейс IDataBound, но на этот раз EditBox не будет предоставлять методы IDataBound внешнему миру. Этот интерфейс нужен ему скорее для собственных нужд, или, возможно, программист просто не хочет засорять пространство имен класса многочисленными методами, которые обычно не используются клиентом. Чтобы скрыть член реализованного интерфейса, нужно лишь удалить модификатор доступа члена public и квалифицировать имя члена именем интерфейса:

```
using System;

public interface IDataBound
{
  void Bind();
}

public class EditBox: IDataBound
{
  //реализация IDataBound
  void IDataBound.Bind()
  {
    Console.WriteLine("Связывание с данными... ");
  }
}

class NameHidding1App
{
  public static void Main()
  {
    EditBox edit = new EditBox();
    Console.WriteLine("Вызов метода EditBox.Bind()");

    // Ошибка
    // EditBox больше не содержит в себе метод Bind
    edit.Bind();

    Console.WriteLine();

    IDataBound bound = (IDataBound)edit;
    Console.WriteLine("Вызов метода (IDataBound)EditBox.Bind()");
    bound.Bind();
  }
}
```

Этот код не будет компилироваться, так как имя члена Bind более не является частью класса EditBox. Поэтому данная методика позволяет вам удалять член из пространства имен класса, в то же время разрешая явный доступ к нему с помощью операции приведения.

При сокрытии члена вы не можете применять модификатор доступа. При попытке использования модификатора доступа с членом реализованного интерфейса вы получите ошибку периода компиляции. Может, это покажется странным, но поймите, что общая причина, заставляющая скрыть что-то,— желание сделать эту сущность невидимой за пределами текущего класса. Так как модификаторы доступа существуют лишь для определения уровня видимости за пределами базового класса, при сокрытии имен они не имеют смысла.

Избежание неоднозначности имен

Одна из главных причин, по которой C# не поддерживает множественное наследование,— проблема конфликта имен, результатом которой является неоднозначность имен. Хотя C# не поддерживает множественное наследование на уровне объектов (создание производных классов), он поддерживает наследование от одного класса и дополнительную реализацию нескольких интерфейсов. Однако за дополнительные возможности приходится расплачиваться конфликтами имен.

Ниже интерфейсы ISerializable и IDataStore поддерживают чтение и хранение данных в разных форматах: в двоичной форме в виде объектов для хранения на диске и для хранения в БД. Проблема в том, что оба содержат методы с именем SaveData:

```
using System;

namespace NameCollisions
{
  interface ISerializable
  {
    void SaveData();
  }
  interface IDataStore
  {
    void SaveData();
  }

  class Test: ISerializable, IDataStore
  {
    public void SaveData()
    {
      Console.WriteLine("Test.SaveData() вызван");
    }
  }

  class NameCollisionsApp
  {
    public static void Main()
    {
      Test test = new Test ();

      Console.WriteLine("Вызов метода Test.SaveData()");
      test.SaveData();
    }
  }
}
```

Независимо от того, компилируется ли этот код, у вас возникнет проблема, так как поведение в результате вызова метода SaveData будет неопределенным для программиста, пытающегося задействовать этот класс. Получите ли вы метод SaveData, который последовательно записывает данные объекта на диск, или метод SaveData, который сохраняет их в БД? В дополнение взгляните на такой код:

```
using System;

namespace NameCollisions
{
  interface ISerializable
  {
    void SaveData();
  }

  interface IDataStore
  {
    void SaveData();
  }

  class Test: ISerializable, IDataStore
  {
    public void SaveData()
    {
      Console.WriteLine("Test.SaveData() вызван");
    }
  }

  class NameCollisionsApp
  {
    public static void Main()
    {
      Test test = new Test();

      if (test is ISerializable)
      {
        //Console.WriteLine("Вызов метода Test.SaveData()");
        Console.WriteLine("Интерфейс ISerializable реализуется");
        //test.SaveData();
      }
      if(test is IDataStore)
      {
        //Console.WriteLine("Вызов метода Test.SaveData()");
        Console.WriteLine("Интерфейс IDataStore реализуется");
        //test.SaveData();
      }
    }
  }
}
```

В этом примере оператор is успешно выполняется в обоих интерфейсах, а значит, реализованы оба интерфейса, хотя мы знаем, что это не так! При компиляции данного примера компилятор даже выдаст предупреждения:

(29,7): warning CS0183: The given expression is always of the provided ('NameCoUisions.ISerializable') type

(35,7): warning CS0183: The given expression is always of the provided ('NameCollisions.IDataStore') type

Проблема в том, что в классе реализованы или сериализованная версия метода Bind, или версия, использующая БД (но не обе сразу). Однако клиент получит положительный результат при проверке на реализацию обеих версий этого интерфейса. При попытке задействовать интерфейс, который на самом деле не реализован, результат непредсказуем.

Чтобы решить эту проблему, можно обратиться к явной квалификации имени члена: уберем модификатор доступа и поставим перед именем члена (в данном случае перед SaveData) имя интерфейса:

```
using System;

namespace NameCollisions
{
  interface ISerializable
  {
    void SaveData();
  }

  interface IDataStore
  {
    void SaveData();
  }

  class Test: ISerializable, IDataStore
  {
    void ISerializable.SaveData()
    {
      Console.WriteLine("Test.ISerializable.SaveData() вызван");
    }
    void IDataStore.SaveData()
    {
      Console.WriteLine("Test.IDataStore.SaveData() вызван");
    }
  }

  class NameCollisionsApp
  {
    public static void Main()
```

```
    {
      Test test = new Test();

      if (test is ISerializable)
      {
        //Console.WriteLine("Вызов метода Test.SaveData()");
        Console.WriteLine("Интерфейс ISerializable реализуется");
        ((ISerializable)test).SaveData();

      !

      Console.WriteLine();

      if (test is IDataStore)
      {
        //Console.WriteLine("Вызов метода Test.SaveData()");
        Console.WriteLine("Интерфейс IDataStore реализуется");
        ((IDataStore)test).SaveData();
      }
    }
  }
}
```

Теперь можно сказать однозначно, какой метод будет вызван. Оба метода реализованы с полностью квалифицированными именами, а приложение выдает именно тот результат, которого вы ожидаете:

Интерфейс ISerializable реализуется
Test.ISerializable.SaveData() вызван

Интерфейс IDataStore реализуется
Test.IDataStore.SaveData() вызван

РОЛЬ ИНТЕРФЕЙСОВ В НАСЛЕДОВАНИИ

С интерфейсами и наследованием связаны две распространенные проблемы. Первая, проиллюстрированная приведенным ниже кодом, связана с созданием производного класса, содержащего метод, чье имя идентично имени метода интерфейса, который должен быть реализован классом:

```
using System;

public interface IDataBound
{
  void Serialize!);
}

public class Control
```

```
{
  public void Serialize!)
  {
    Console.WriteLine("Метод Control.Serialize вызван"];
  )
}

public class EditBox: Control, IDataBound
{
}

class InterfaceInhApp
{
  public static void Main()
  f
    EditBox edit = new EditBox!);
    edit.Serialize();
  }
}
```

Как вы знаете, чтобы реализовать интерфейс, нужно предоставить определение каждого члена в определении интерфейса. Однако в предыдущем примере мы этого не сделали, а код все равно компилируется! Причина в том, что компилятор C# ищет метод Serialize, реализованный в классе EditBox, и находит его. Однако компилятор неверно определяет, что это реализованный метод. Метод Serialize, найденный компилятором, унаследован от класса Control методом, но не является настоящей реализацией метода IDataBound.Serialize. Поэтому, хоть он и компилируется, код не будет функционировать, как задумано, в чем мы убедимся позже.

Теперь внесем дополнительную интригу. Следующий код сначала проверяет оператором as, реализован ли интерфейс, затем пытается вызвать реализованный метод Serialize. Этот код компилируется и работает. Однако, как мы знаем, в классе EditBox метод Serialize на самом деле не реализован из-за наследования IDataBound. В EditBox уже есть метод Serialize (унаследованный) от класса Control. Это значит, что клиент, по всей вероятности, не получит ожидаемых результатов.

```
using System;

public interface IDataBound
{
  void Serialize();
}

public class Control
{
  public void Serialize()
```

```
  {
    Console.WriteLine("Метод Control.Serialize вызван");
  }
}

public class EditBox: Control, IDataBound
{
}

class InterfaceInhApp
{
  public static void Main()
  {
    EditBox edit = new EditBox();

    IDataBound bound = edit as IDataBound;
    if(bound != null)
    {
      Console.WriteLine("Интерфейс IDataBound поддерживается");
      bound.Serialize();
    }
    else
    {
      Console.WriteLine("Интерфейс IDataBound не поддерживается");
    }
  }
}
```

Возникновение другой потенциальной проблемы ожидается, когда у производного класса есть метод с тем же именем, что и у реализации метода интерфейса из базового класса. Давайте рассмотрим этот код:

```
using System;

public interface ITest
{
  void Foo();
}

class Base: ITest
{
  public void Foo()
  {
    Console.WriteLine(
      "Реализация метода Foo интерфейса ITest в базовом классе");
  }
}

class Derived: Base
```

```
{
  public new void Foo()
  {
    Console.WriteLine("Derived.Foo");
  }
}

class InterfaceInhApp
{
  public static void Main()
  {
    Derived der = new Derived!);
    der.Foo();

    ITest test = (ITest)der;
    test.Foo();
  ]
}
```

В результате выполнения этот код выводит информацию:

Derived.Foo

Реализация метода Foo интерфейса ITest в базовом классе

В этой ситуации в классе Base реализован интерфейс ITest и его метод Foo. Однако производный от Base класс Derived реализует для этого класса новый метод Foo. Какой из методов Foo будет вызван? Это зависит от имеющейся у вас ссылки. Если есть ссылка на объект Derived, вызывается его метод Foo. Это так, даже несмотря на то, что у объекта der есть унаследованная реализация ITest.Foo. В период выполнения будет исполнен Derived.Foo, так как ключевым словом new задана подмена унаследованного метода.

Однако когда вы явно выполняете приведение объекта der к интерфейсу ITest, компилятор разрешает реализацию интерфейса. У класса Derived есть метод с тем же именем, но это не тот метод, что ищет компилятор. Когда вы приводите объект к интерфейсу, компилятор проходит по дереву наследования, пока не найдет класс, содержащий в своем базовом списке интерфейс. Именно поэтому в результате выполнения последних двух строк кода метода Main вызывается метод Foo, реализованный в ITest.

Хочется надеяться, что некоторые из этих потенциальных ловушек, включая конфликты имен и наследование интерфейсов, убедили вас следовать рекомендации: всегда приводить объекты к интерфейсу, член которого вы пытаетесь использовать.

КОМБИНИРОВАНИЕ ИНТЕРФЕЙСОВ

Еще одна мощная функция C# — возможность комбинирования двух или более интерфейсов, в результате чего класс должен реализовать только результат комбинирования. Допустим, вы хотите создать новый класс

TreeView, реализующий два интерфейса: IDragDrop и ISerializable. Поскольку резонно предполагать, что и другим элементам управления, таким как ListView и ListBox, понадобится скомбинировать эти функции, вам, возможно, захочется скомбинировать интерфейсы IDragDrop и ISerializable в единое целое:

```
using System;

public class Control
{
}

public interface IDrugDrop
{
  void Drag();
  void Drop() ;
}

public interface ISerializable
{
  void Serialize ();
}
public interface ICombo:IDrugDrop, ISerializable
{
  //этот интерфейс объединяет в себе
  //два других интерфейса
}

public class MyTreeView: Control, ICombo
{
  public void Drag()
  {
    Console.WriteLine("Вызов метода MyTreeView.Drag");
  }
  public void Drop ()
  {
    Console.WriteLine("Вызов метода MyTreeView.Drop");
  }
  public void Serialize()
  {
    Console.WriteLine("Вызов метода MyTreeView.Serialize");
  }
}

class CombiningApp
{
  public static void Main()
```

```
  {
    MyTreeView tree = new MyTreeView();
    tree.Drag();
    tree.Drop();
    tree.Serialize();
  }
}
```

Комбинируя интерфейсы, вы не только упростите возможность использование связанных интерфейсов, но при необходимости сможете добавлять к новому «композитному» интерфейсу дополнительные методы.

16. ДЕЛЕГАТЫ И ОБРАБОТЧИКИ СОБЫТИЙ

Одно из полезных нововведений в C# — *делегаты* (delegates). Их назначение, по сути, совпадает с указателями функций в C++, но делегаты являются управляемыми объектами и привязаны к типам. Это значит, что исполняющая среда (runtime) гарантирует, что делегат указывает на допустимый объект, а это в свою очередь означает получение всех достоинств указателей функций без связанных с этим опасностей, таких как применение недопустимых адресов или разрушение памяти других объектов. В этой главе мы рассмотрим делегаты в сравнении с интерфейсами, их синтаксис и проблемы применения. Мы также увидим несколько примеров использования делегатов с функциями обратного вызова и асинхронными обработчиками событий.

Из главы 15 вы узнали, как определяются и реализуются интерфейсы, а также то, что с концептуальной точки зрения интерфейсы — это связки между двумя различными частями кода. Но при этом интерфейсы во многом напоминают классы, так как объявляются в период компиляции и могут включать методы, свойства, индексаторы и события. Что касается делегата, то он ссылается на единственный метод и определяется в период выполнения. В C# две основных области применения делегатов: методы обратного вызова и обработчики событий.

МЕТОДЫ ОБРАТНОГО ВЫЗОВА

Методы обратного вызова повсеместно используются в Windows для передачи указателя функции другой функции, чтобы последняя могла вызвать первую (через переданный ей указатель). Так, функция Win32 API EnumWindows перечисляет все окна верхнего уровня на экране и для каждого окна вызывает переданную ей функцию. Обратные вызовы применяют по-разному, но наиболее распространены два случая.

Асинхронная обработка. Методы обратного вызова используют при асинхронной обработке, когда вызванному коду требуется значительное время для обработки запроса. Обычно сценарий таков. Клиентский код вызывает метод, передавая ему метод обратного вызова. Вызванный метод начинает работу в своем потоке и сразу возвращает управление. Запущенный поток затем выполняет основную работу, при необходимости обращаясь к функции обратного вызова. Очевидное достоинство такого подхода в том, что клиент продолжает работу, не блокируясь на потенциально длительное время, которое требуется для синхронного вызова.

Введение дополнительного кода в код класса. Другой распространенный способ применения методов обратного вызова имеет место, когда класс позволяет клиенту указать метод для дополнительной нестандартной обработки. Например, в классе Windows Listbox можно указать нисходящий или восходящий порядок сортировки элементов. Кроме некоторых других базовых возможностей для сортировки, этот класс на самом деле не дает полной свободы действий и остается общим классом. Но при этом Listbox позволяет указывать для сортировки функцию обратного вызова. Таким образом, Listbox для сортировки вызывает функцию обратного вызова, и ваш код может выполнять нужные нестандартные действия.

Рассмотрим пример определения и применения делегата. Предположим, у нас есть класс менеджера базы данных, который отслеживает все активные соединения с БД и предоставляет метод перечисления этих соединений. Если допустить, что менеджер БД находится на удаленной машине, правильно будет сделать метод асинхронным, позволив клиенту предоставлять метод обратного вызова. Заметьте: в реальном приложении вам следовало бы задействовать многопоточность, чтобы добиться подлинной асинхронности. Но для упрощения примера и с учетом того, что мы не рассматривали многопоточность, не будем ее применять.

Для начала определим два главных класса: DBManager и DBConnection.

```
class DBConnection
{ ... }
class DBManager
{
  static DBConnection[] activeConnections;
  public delegate void EnumConnectionsCallback(
    DBConnection connection);
  public static void EnumConnections (
    EnumConnectionsCaliback callback)
  {
    foreach (DBConnection connection in activeConnections)
    {
      callback(connection);
    }
  }
}
```

Метод EnumConnectionsCallback является делегатом, что определяется ключевым словом delegate в начале сигнатуры метода. Как видите, этот делегат возвращает void и принимает единственный аргумент — объект DBConnection. Метод EnumConnections в соответствии с его определением принимает единственный аргумент — метод EnumConnectionsCallback. Чтобы вызвать метод DBManager.EnumConnections, нам нужно лишь передать ему экземпляр делегата DBManager.EnumConnectionsCallback.

Для создания экземпляра делегата нужно применить new, передав ему имя метода, имеющего ту же сигнатуру, что и у делегата. Вот пример:

```
DBManager.EnumConnectionsCallback myCallback =
  new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);
DBManager. EnumConnections(myCallback);
```

Заметьте, что это можно скомбинировать в единый вызов:

```
DBManager.EnumConnections(new
DBManager.EnumConnectionsCallback(ActiveConnectionsCallback));
```

Вот и все, что касается базового синтаксиса делегатов. Теперь посмотрим на законченный пример:

```
using System;
class DBConnection
{
  public DBConnection(string name)
  f
    this.name = name;
  }
  protected string Name;
  public string name
  {
    get
    {
      return this.Name;
    }
    set
    {
      this.Name = value;
    }
  }
}

class DBManager
{
  static DBConnection[] activeConnections;
  public void AddConnections()
  \
    activeConnections = new DBConnection[5];
    for (int i = 0; i < 5; i++)
    {
      activeConnections[i] =
        new DBConnection("DBConnection " + (i+ 1));
    }
  }
  public delegate void EnumConnectionsCallback(DBConnection connection);
  public static void EnumConnections(EnumConnectionsCallback callback)
```

```
  {
    foreach (DBConnection connection in activeConnections)
    i
      callback(connection);
    }
  }
}
class DelegatelApp
i
  public static void ActiveConnectionsCallback(DBConnection connection)
  t
    Console. WriteLine ( "Callback функция вызвана для {0}", connection.name);
  }
  public static void Main()
  {
    DBManager dbMgr = new DBManager();
    dbMgr.AddConnections();
    DBManager.EnumConnectionsCallback myCallback = new
    DBManager.EnumConnectionsCallback ( ActiveConnectionsCallback);
    DBManager.EnumConnections(myCallback);
  ]
}
```

После компиляции и запуска этого приложения мы получим такие результаты:

Callback функция вызвана для DBConnection 1
Callback функция вызвана для DBConnection 2
Callback функция вызвана для DBConnection 3
Callback функция вызвана для DBConnection 4
Callback функция вызвана для DBConnection 5

ДЕЛЕГАТЫ КАК СТАТИЧЕСКИЕ ЧЛЕНЫ

Довольно неуклюжее решение создавать экземпляр делегата при каждом его применении, но в C# можно определять метод, который используется при создании делегата, как статический член класса. Ниже приведен пример с применением такого подхода. Заметьте, что теперь делегат определен как статический член класса myCallback и может использоваться в методе Main, так что клиенту нет нужды создавать экземплярделегата:

```
using System;

class DBConnection
{
  public DBConnection(string name)
```

```
  {
    this.name = name;
  }

  protected string Name;
  public string name
  {
    get
    {
      return this.Name;
    }
    set
    {
      this.Name = value;
    }
  }
}

class DBManager
{
  static DBConnection[] activeConnections;
  public void AddConnections()
  {
    activeConnections = new DBConnection[5];
    for (int i = 0; i < 5; i++)
    {
      activeConnections[i] = new DBConnection("DBConnection " + (i + 1));
    }
  }

  public delegate void EnumConnectionsCallback(DBConnection connection);
  public static void EnumConnections(EnumConnectionsCallback callback)
  {
    foreach (DBConnection connection in activeConnections)
    {
      callback(connection);
    }
  }
}

class Delegate2App
{
  public static DBManager.EnumConnectionsCallback rayCallback =
    new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

  public static void ActiveConnectionsCallback(DBConnection connection)
```

```
  {
    Console.WriteLine("Callback функция вызвана для {0}", connection.name);
  }

  public static void Main()
  {
    DBManager dbMgr = new DBManager();
    dbMgr.AddConnections();

    DBManager.EnumConnections(myCallback);
  }
}
```

Общим правилом именования делегатов является добавление слова Callback к имени метода, принимающего делегат в качестве аргумента. Можно поошибке использовать имя этого метода вместо имени делегата. При этом компилятор уведомит, что вы указали метод там, где ожидается класс. Получив такую ошибку, помните: проблема в том, что вы указали метод вместо делегата.

В двух примерах, которые мы рассмотрели, делегаты создаются независимо от того, будут ли они когда-либо использоваться. В рассмотренных вариантах ничего плохого в этом нет, поскольку известно, что они будут вызываться. Но в целом при определении делегатов важно решить, когда их создавать. Скажем, может случиться, что создание делегатов потребует немало времени и затрат. В таких случаях, когда вы знаете, что клиент обычно не обращается к данному методу обратного вызова, можно отложить создание делегата до момента, пока он действительно не потребуется, заключив создание его экземпляра в оболочку свойства. Это иллюстрирует следующий измененный класс DBManager, в котором для создания экземпляра делегата используется неизменяемое свойство (поскольку для него представлен только метод-получатель);

```
using System;

class DBConnection
{
  public DBConnection(string name)
  {
    this.name = name;
  }

  protected string Name;
  public string name
  {
    get
    i
      return this.Name;
    }
```

```
    set
    {
      this.Name = value;
    }
  }
}

class DBManager
{
  static DBConnection[] activeConnections;
  public void AddConnections()
  {
    activeConnections = new DBConnection[5];
    for (int i = 0; i < 5; i++)
    {
      activeConnections[i] = new DBConnection("DBConnection " + (i + 1));
    }
  }

  public delegate void EnumConnectionsCallback(DBConnection connection);
  public static void EnumConnections(EnumConnectionsCallback callback)
  {
    foreach (DBConnection connection in activeConnections)
    {
      callback(connection);
    }
  }
}

class Delegate3App
{
  public DBManager.EnumConnectionsCallback myCallback
  {
    get
    {
      return new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);
    }
  }

  public static void ActiveConnectionsCallback(DBConnection connection)
  {
    Console.WriteLine("Callback функция вызвана для " + connection.name);
  }

  public static void Main()
  {
    Delegate3App app = new Delegate3App();
```

```
    DBManager dbMgr = new DBManager();
    dbMgr.AddConnections();

    DBManager.EnumConnections(app.myCallback);
  }
}
```

СОСТАВНЫЕ ДЕЛЕГАТЫ

Объединение делегатов — создание одного делегата из нескольких — одна из тех возможностей, которая поначалу не кажется такой уж полезной, но если вы столкнетесь с такой потребностью, то будете признательны команде разработчиков C# за то, что они это предусмотрели. Рассмотрим некоторые примеры, когда объединение делегатов может быть полезно. В первом примере мы имеем дистрибьюторскую систему и класс, просматривающий все наименования товара на данном складе, вызывая метод обратного вызова для каждого наименования, запасов которого менее 50 единиц. В реальном дистрибьюторском приложении формула должна учитывать не только наличные запасы, но также заказанные и находящиеся «в пути». Но возьмем простой пример: если наличие на складе менее 50 единиц, возникает исключение.

Фокус в том, что мы хотим разделить методы, которые вызываются, если запасы ниже допустимых: нам нужно запротоколировать сам факт и, кроме того, послать письмо по электронной почте менеджеру по закупкам. Итак, составим делегат из нескольких других:

```
using System;
using System.Threading;

class Part
{
  public Part(string sku)
  {
    this.Sku = sku;

    Random r = new Random(DateTime.Now.Millisecond);
    double d = r.NextDouble() * 100;

    this.OnHand = (int)d;
  }

  protected string Sku;
  public string sku
  {
    get
```

```
    {
      return this.Sku;
    i
    set
    {
      this.Sku = value;
    }
  }

  protected int OnHand;
  public int onhand
  {
    get
    {
      return this.OnHand;
    }
    set
    {
      this.OnHand = value;
    }
  i
)

class InventoryManager
{
  protected const int MIN_ONHAND = 50;

  public Part[] parts;
  public InventoryManager()
  {
    parts = new Part[5];

    for (int i = 0; i < 5; i++)

      Part part = new Part("Товар " + (i + 1));

      Thread.Sleep(10);

      parts[i] = part;                                          ",
      Console.WriteLine("Добавление товара '{0}' в наличии = {1}'
    }   part.sku, part.onhand);
  }

public delegate void OutOfStockExceptionMethod(Part part);
public void ProcessInventory(OutOfStockExceptionMethod exception)
{
  Console.WriteLine("\Инвентаризация товара...");
```

```
    foreach (Part part in parts)
    i
      if (part.onhand < MIN_ONHAND)
      {
        Console.WriteLine("{0}: имеется в наличии {1}, " +
          "это меньше, чем необходимо ({2})",
          part.sku, part.onhand, MIN_ONHAND);

        exception(part);
      }
    ]
  }
}


class CompositeDelegate1App
{
  public static void LogEvent(Part part)
  {
    Console.WriteLine("\tпротоколирование события ... ");
  )

  public static void EmailPurchasingMgr(Part part)
  {
    Console.WriteLine("\tуведомление менеджера по e-mail...");

  }

  public static void Main()
  {

    InventoryManager mgr = new InventoryManager();

    InventoryManager.OutOfStockExceptionMethod LogEventCallback =
      new InventoryManager.OutOfStockExceptionMethod(LogEvent);

    InventoryManager.OutOfStockExceptionMethod EmailPurchasingMgrCallback =
      new InventoryManager.OutOfStockExceptionMethod(EmailPurchasingMgr);

    InventoryManager.OutOfStockExceptionMethod OnHandExceptionEventsCallback =
      EmailPurchasingMgrCallback + LogEventCallback;

    mgr.ProcessInventory(OnHandExceptionEventsCallback);
  }
}
```

В результате выполнения мы увидим примерно такой результат:

Добавление товара 'Товар V в наличии = 0
Добавление товара 'Товар 2' в наличии = 36

Добавление товара 'Товар 3' в наличии = 29
Добавление товара 'Товар 4' в наличии = 12
Добавление товара 'Товар 5' в наличии = 48

Инвентаризация товара...
Товар 1: имеется в наличии 0, это меньше, чем необходимо (50)
уведомление менеджера по e-mail...
протоколирование события...
Товар 2: имеется в наличии 36, это меньше, чем необходимо (50)
уведомление менеджера по e-mail...
протоколирование события...
Товар 3: имеется в наличии 29, это меньше, чем необходимо (50)
уведомление менеджера по e-mail...
протоколирование события...
Товар 4: имеется в наличии 12, это меньше, чем необходимо (50)
уведомление менеджера по e-mail...
протоколирование события...
Товар 5: имеется в наличии 48, это меньше, чем необходимо (50)
уведомление менеджера по e-mail...
протоколирование события...

Давайте рассмотрим программу подробнее. Вначале объявляется класс Part:

```
class Part
{
  public Part(string sku)
  {
    this.Sku = sku;

    Random r = new Random(DateTime.Now.Millisecond);
    double d = r.NextDouble() * 100;

    this.OnHand = (int)d;
  }

  protected string Sku;
  {...}
  protected int OnHand;
  {...}
}
```

Этот класс предназначен для хранения информации о количестве товара, расположенного на складе, и наименовании товара. В конструкторе класса передается наименование товара и устанавливается его количество. Количество товара устанавливается при помощи генерации случайных чисел. Класс Random генерирует в произвольный момент времени произвольное число. Функция Random.NextDouble () возвращает значение от 0 до 1. При записи этого значения в переменную d мы умножаем воз-

вращаемое значение на 100. При этом мы получаем число в пределах от 0 до 100. Это сгенерированное произвольным образом значение устанавливается свойству OnHand.

Класс имеет два свойства sku и onhand.

Далее следует описание класса InventoryManager:

```
class InventoryManager
{
  protected const int MIN_ONHAND = 50;

  public Part[] parts;
  public InventoryManager()
  {
    parts = new Part[5];
    for (int i = 0; i < 5; i++)
    {
      Part part = new Part ("Товар " + (i + 1));

      Thread.Sleep(10);

      parts[i] = part;
      Console.WriteLine("Добавление товара '{0}' в наличии = {1}",
        part.sku, part.onhand);
    }
  }

  public delegate void OutOfStockExceptionMethod(Part part);
  public void ProcessInventory(OutOfStockExceptionMethod exception)
  {
    Console.WriteLine("\nProcessing inventory...");
    foreach (Part part in parts)
    {
      if (part.onhand < MIN_ONHAND)
      {
        Console.WriteLine("{0}: имеется в налички {1}, " +
          "это меньше, чем необходимо (f2})",
          part.sku, part.onhand, MIN_ONHAND);
        exception(part);
      }
    }
  }
}
```

Константная переменная MIN_ONHAND означает критическое количество товара. Значит, если количество товара на складе менее этого значения, то необходимо предпринять дополнительные действия для решения возникшей проблемы. В классе InventoryManager объявлен массив объектов Part:

```
public Part[] parts
```

Конструктор инициализирует этот массив значениями. Инициализация производится строкой «Товар» + порядковый номер элемента в массиве. В цикле инициализации также вызывается функция

```
Thread.Sleep(10);
```

которая заставляет программу приостановиться на 10 миллисекунд. Это необходимо для того, чтобы значения, генерируемые функцией Random, имели больший разброс.

Далее в классе InventoryManager объявляется делегат OutOfStockExceptionMethod, принимающий параметр типа Part.

```
public delegate void OutOfStockExceptionMethod(Part part);
```

Функция ProcessInventory производит инвентаризацию товара на складе и в случае возникновения исключения вызывает делегат. Функция ProcessInventory просматривает все имеющиеся на складе товары и делает анализ количества каждого. Если количество какого-либо товара менее 50, то вызывается метод exception. Метод exception является делегатом, который передается функции ProcessInventory вызывающим методом.

Класс самого приложения содержит объявление двух делегатов, объединенных в один составной делегат.

```
class CompositeDelegateApp
{
  public static void LogEvent(Part part)
  {
    Console.WriteLine("\tпротоколированиесобытия. . . ");
  }

  public static void EmailPurchasingMgr(Part part)
  {
    Console.WriteLine ("\tуведомление менеджера по e-mail ...");
  }

  public static void Main()
  {
    InventoryManager mgr = new InventoryManager();

    InventoryManager.OutOfStockExceptionMethod LogEventCallback =
      new InventoryManager.OutOfStockExceptionMethod(LogEvent);

    InventoryManager.OutOfStockExceptionMethod EmailPurchasingMgrCallback =
      new InventoryManager.OutOfStockExceptionMethod(EmailPurchasingMgr);

    InventoryManager.OutOfStockExceptionMethod OnHandExceptionEventsCallback =
      EmailPurchasingMgrCallback + LogEventCallback;

    mgr.ProcessInventory(OnHandExceptionEventsCallback);
  }
}
```

Первый делегат LogEvent предназначен для создания протокола. Второй делегат выполняет функции отсылки письма менеджеру с уведомлением.

В функции Main создаются экземпляры делегатов с именами LogEventCallback и EmailPurchasingMgrCallback. Затем создается экземпляр составного делегата с именем OnHandExceptionEventsCallback, который инициализируется сразу двумя простыми делегатами: EmailPurchasingMgrCallback+ LogEventCallback.

Таким образом, эта возможность языка позволяет динамически определять, какие методы нужно включать в метод обратного вызова и объединять их в составной делегат. Исполняющая среда распознает, что эти методы нужно вызвать последовательно. Кроме того, вы можете убрать любой делегат из составного оператором «минус».

Тот факт, что эти методы вызываются последовательно, заставляет спросить: почему просто не связать методы в цепочку, чтобы каждый метод вызывал последующий? В нашем примере, где у нас всего два метода, которые вызываются всегда одновременно, это сделать можно. Но усложним пример. Допустим, у нас несколько магазинов, расположенных в разных местах, и каждый сам решает, какие методы вызывать. К примеру, на территории одного магазина находится общий товарный склад, и здесь нужно запротоколировать событие и сообщить менеджеру о закупках, а в других магазинах — запротоколировать событие и сообщить управляющему магазина.

Такое требование легко реализовать, динамически создавая составной делегат на основе информации о конкретном местоположении магазина. Без делегатов нам пришлось бы написать метод, который не только определял бы, какие методы вызывать, но и отслеживал, какие методы уже вызывались, а какие еще нужно вызвать. Смотрите, как делегаты упрощают эту потенциально сложную операцию:

```
using System;
using System.Threading;

class Part
{
  public Part(string sku)
  {
    this.Sku = sku;

    Random r = new Random(DateTime.Now.Millisecond);
    double d = r.NextDouble() * 100;

    this.OnHand = (int)d;
  }

  protected string Sku;
  public string sku
```

```
  {
    get
    {
      return this.Sku;
    }
    set
    {
      this.Sku = value;
    }
  }

  protected int OnHand;
  public int onhand
  {
    get
    {
      return this.OnHand;
    }
    set
    {
      this.OnHand = value;
    }
  }
}

class InventoryManager
{
  protected const int MIN_ONHAND = 50;

  public Part [ ] parts;
  public InventoryManager()
  {
    parts = new Part[5];
    for (int i = 0; i < 5; i++)
    {
      Part part = new Part("Товар " + (i + 1));

      Thread.Sleep(10);

      parts[i] = part;
      Console.WriteLine("Добавление товара '{0}' в наличии = {1}",
        part.sku, part.onhand);
    }
  }

  public delegate void OutOfStockExceptionMethod(Part part);
```

```
  public void ProcessInventory(OutOfStockExceptionMethod exception)
  {
    Console.WriteLine("\Инвентаризация товара...");
    foreach (Part part in parts)
    {
      if (part.onhand < MIN_ONHAND)
      {
        Console.WriteLine("{0}: имеется в наличии {1}, " +
          "это меньше чем необходимо ({2})",
          part.sku, part.onhand, MIN_ONHAND) ;

        exception(part);
      }
    }
  }
}

class CompositeDelegate2App
{
  public static void LogEvent(Part part)
  {
    Console.WriteLine("\tпротоколирование события... ") ;
  }

  public static void EmailPurchasingMgr(Part part)
  {
    Console.WriteLine ("\tуведомление менеджера магазина... ") ;
  }

  public static void EmailStoreMgr(Part part)
  {
    Console.WriteLine ("\tуведомление менеджера по закупкам... ");
  }

  public static void Main()
  {
    InventoryManager mgr = new InventoryManager();

    InventoryManager.OutOfStockExceptionMethod[] exceptionMethods
      = new InventoryManager.OutOfStockExceptionMethod[3];

    exceptionMethods[0] = new
    InventoryManager.OutOfStockExceptionMethod(LogEvent);
    exceptionMethods[1] = new
    InventoryManager.OutOfStockExceptionMethod(EmailPurchasingMgr);
    exceptionMethods[2] = new
    InventoryManager.OutOfStockExceptionMethod(EmailStoreMgr);
```

```
    int location = 2;

    InventoryManager.OutOfStockExceptionMethod compositeDelegate;

    if (location == 2)
    {
      compositeDelegate = exceptionMethods[0] + exceptionMethods[1];
    }
    else
    {
      compositeDelegate = exceptionMethods[0] + exceptionMethods[2];
    ;

    mgr.ProcessInventory(compositeDelegate);
  }
}
```

Теперь при компиляции и выполнении этого приложения результаты будут отличаться в зависимости от значения переменной location. Например, если вы зададите значение переменной location = 2, то сообщения будут приходить менеджеру магазина:

Добавление товара 'Товар 1' в наличии = 81
Добавление товара 'Товар 2' в наличии = 17
Добавление товара 'Товар 3' в наличии = 53
Добавление товара 'Товар 4' в наличии = 36
Добавление товара 'Товар 5' в наличии = 72

Инвентаризация товара...
Товар 2: имеется в наличии 17, это меньше чем необходимо (50)
протоколирование события...
уведомление менеджера магазина...
Товар 4: имеется в наличии 36, это меньше чем необходимо (50)
протоколирование события...
уведомление менеджера магазина...

Если же вы зададите значение переменной равным 1, то сообщения будут приходить менеджеру по закупкам:

Добавление товара 'Товар 1' в наличии = 52
Добавление товара 'Товар 2' в наличии = 88
Добавление товара 'Товар 3' в наличии = 81
Добавление товара 'Товар 4' в наличии = 65
Добавление товара 'Товар 5' в наличии = 1

Инвентаризация товара...
Товар 5: имеется в наличии 1, это меньше чем необходимо (50)
протоколирование события...
уведомление менеджера по закупкам...

ОПРЕДЕЛЕНИЕ СОБЫТИЙ С ПОМОЩЬЮ ДЕЛЕГАТОВ

Практически во всех Windows-приложениях требуется асинхронная обработка событий. Некоторые из этих событий связаны с самой ОС, например, когда Windows посылает сообщения в очередь сообщений приложения при том или ином взаимодействии пользователя с приложением. Другие больше связаны с предметной областью, например, когда нужно распечатать счет при обновлении заказа.

Работа с событиями в C# соответствует модели «издатель — подписчик», где класс публикует событие, которое он может инициировать, и любые классы могут подписаться на это событие. При инициации события исполняющая среда следит за тем, чтобы уведомить всех подписчиков о возникновении события.

Метод, вызываемый при возникновении события, определяется делегатом. Однако нужно помнить об ограничениях, которые налагаются на делегаты, используемые для этих целей. Во-первых, нужно, чтобы такой делегат принимал два аргумента. Во-вторых, эти аргументы всегда должны представлять два объекта: тот, что инициировал событие (издатель), и информационный объект события, который должен быть производным от класса EventArgs .NET Framework.

Скажем, мы хотим отслеживать изменения объемов запасов. Мы создаем класс Inventor/Manager, который будет всегда использоваться при обновлении запасов. Этот класс должен публиковать события, возникающие при всяком изменении запасов вследствие закупок, продаж и других причин. Тогда любой класс, которому нужно отслеживать изменения, должен подписаться на эти события. Вот как это делается на C# при помощи делегатов и событий:

```
using System;

class InventoryChangeEventArgs: EventArgs
{
  public InventoryChangeEventArgs(string sku, int change)
  ;
    this.sku = sku;
    this.change = change;
  }

  string sku;
  public string Sku
  {
    get
    i
    } return sku;
  }
```

```
  int change;
  public int Change
  {
    get
    {
      return change;
    }
  }
}

class InventoryManager // издатель
{
  public delegate void InventoryChangeEventHandler(
    object source, InventoryChangeEventArgs e);
  public event InventoryChangeEventHandler OnInventoryChangeHandler;

  public void UpdateInventory(string sku, int change)
  {
    if (0 == change)
      return;

    InventoryChangeEventArgs e = new InventoryChangeEventArgs(sku, change);

    if (OnInventoryChangeHandler != null)
      OnInventoryChangeHandler(this, e);
  }
}

class InventoryWatcher // подписчик
{
  public InventoryWatcher(InventoryManager inventoryManager)
  {
    this.inventoryManager = inventoryManager;
    inventoryManager.OnInventoryChangeHandler
      += new InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
  }
  void OnInventoryChange(object source, InventoryChangeEventArgs e)
  {
    int change = e.Change;
    Console.WriteLine("Количество товара '{0}' было {1} на {2} единиц",
      e.Sku,
      change > 0 ? "увеличено": "уменьшено",
      Math.Abs(e.Change));
  }
  InventoryManager inventoryManager;
}

class Events1App
```

```
{
  public static void Main()
  {
    InventoryManager inventoryManager = new InventoryManager();

    InventoryWatcher inventoryWatch = new InventoryWatcher(inventoryManager);

    inventoryManager.UpdateInventory("Кетчуп Балтимор", -7);
    inventoryManager.UpdateInventory("Масло растительное", 5);
  }
}
```

Первая строка кода — это делегат, вы можете узнать его по определению сигнатуры метода. Как уже говорилось, все делегаты, используемые с событиями, должны быть определены с двумя аргументами: объект-издатель (в данном случае source) и информационный объект события (производный от EventArgs). Во второй строке указано ключевое слово event, член, имеющий тип указанного делегата, и метод (или методы), которые будут вызываться при возникновении события.

UpdateInventory — последний метод класса InventoryManager — вызывается при каждом изменении запасов. Как видите, он создает объект типа InventoryChangeEventArgs, который передается всем подписчикам и описывает возникшее событие.

Теперь рассмотрим еще две строки кода:

```
if (OnInventoryChangeHandler != null)
  OnInventoryChangeHandler(this, e);
```

Условный оператор if проверяет, есть ли подписчики, связанные с методом OnInventoryChangeHandler. Если это так, т. е. OnInventoryChangeHandler не равен null, инициируется событие. Вот и все, что касается издателя. Теперь рассмотрим код подписчика.

Подписчик в нашем случае представлен классом InventoryWatcher. Ему нужно выполнять две простые задачи. Во-первых, он должен указать себя как подписчик, создав новый экземпляр делегата типа InventoryManager.InventoryChangeEventHandler, и добавить этот делегат к событию InventoryManager, OnInventoryChangeHandler, Обратите особое внимание на синтаксис: для добавления себя к списку подписчиков он использует составной оператор присваивания +=, чтобы не уничтожить предыдущих подписчиков.

```
inventoryManager.OnInventoryChangeHandler +=
new InventoryManager.InventoryChangeEventHandler (OnInventoryChange);
```

Единственный аргумент, указываемый здесь,— имя метода, который вызывается при возникновении события.

Последняя задача, которую должен выполнить подписчик,— это реализовать свой обработчик событий. В данном случае обработчик ошибок InventoryWatcher.OnInventoryChange выводит сообщение об изменении запасов с указанием номера позиции.

И, наконец, в коде этого приложения при каждом вызове метода InventoryManager.UpdateInventory создаются экземпляры классов InventoryManager и InventoryWatcher и автоматически инициируется событие, которое приводит к вызову метода InventoryWatcher.OnInventoryChange.

17. ОСОБЫЕ ВОЗМОЖНОСТИ C# И Visual Studio .NET

XML ДОКУМЕНТИРОВАНИЕ КОДА C#

C# имеет особые возможности при создании комментариев к исходному коду. Такой возможностью является XML документирование вашего C# кода. Если вы ознакомитесь с опциями компилятора C#, то заметите наличие параметра «/doc». Используя его, вы можете получить из исходного C# файла XML файл, описывающий его. Полученный файл вы можете обрабатывать как хотите, например, преобразовав в HTML-документацию. Конечно же, существуют правила документирования кода. Основные правила документирования будут рассмотрены в этой главе.

Тэги документирования представлены в следующей таблице:

<c>	<code>	<example>	<exception>	<list>
<para>	<param>	<paramref>	<permission>	<remarks>
<returns>	<see>	<seealso>	<summary>	<value>

<c>

Тэг <c> указывает на то, что текст, заключенный в него, является кодом.

<code>

Тэг <code> имеет то же самое значение, что и тэг <c>, но используется для маркировки нескольких строк.

<example>

Тэг <example> применяется для указания примера использования какого-либо метода или иного элемента вашего модуля.

<exception>

Тэг <exception> используется для комментирования исключений в вашем коде. Для этого параметр тэга cref=«...» должен содержать System.Exception. Компилятор проверит, существует ли исключение, и в случае отсутствия выдаст сообщение об ошибке. В следующем примере демонстрируется использование этого тэга:

```
/// <exception cref="System.Exception"> Это исключение </exception>
class SampleClass: Exception { }
```

Результат обработки тэга <exception> таков: в XML файле, в тэге <member name=«T:SampleClass»>, соответствующем элементу вашего кода, будет находиться тэг <exception> с параметром cref, указываю-

щим имя элемента. Внутри этого тэга будет помещаться ваш комментарий. Например, таким будет результат компилирования последнего примера:

```
<?xml version="1.0"?>
<doc>
 <members>
 <member name="T:SampleClass">
 <exception cref="T:System.Exception"> Это исключение </exception>
 </member>
 </members>
</doc>
```

<list>

Тэг <list> используется для создания таблицы, ненумерованного, нумерованного списка или списка определений. У него есть параметр type, который может принимать три значения: «bullet» — для ненумерованного списка, «number» — для нумерованного списка и «table» — для таблицы.

Внутри <list> могут находиться два вида тэгов: <listheader> и <item>. Каждый из них может содержать внутри себя два тэга — <term> и <description>.

<listheader> используется для описания заголовка таблицы или списка, тэг <item> для указания элемента. В случае таблицы нумерованных или ненумерованных списков внутри тэгов <listheader> и <itera> используется только тэг <term>, в случае списка определений <term> и <description>.

Следующий пример демонстрирует использование <term> для создания списка определений:

```
public class SampleClass {
/// <list type="bullet">
/// <listheader>
/// <term> Фрукты </term>
/// </listheader>
/// <item>
/// <term> Яблоко </term>
/// <description> растет на яблоне </description>
/// </item>
/// <item>
/// <term> Банан </term>
/// <description> растет на пальме </description>
/// </item>
/// <item>
/// <term> Груша </term>
/// <description> растет на груше </description>
/// </item>
/// </list>
public static void Main () { }
}
```

<para>

Тэг <para> используется для обозначения параграфа. Его следует применять внутри таких тэгов, как <return> и <remark>.

<param>

Тэг <param> используется для описания аргументов методов. Его параметр name содержит имя аргуметна метода, в сам же тэг заключено описание аргумента.

```
public class SampleClass {
/// <param name="input"> Содержит строку ввода </param>
public static void SampleMethod (String input) { }
```

<paramref>

Тэг <paramref> показывает, что слово является аргументом метода. Его параметр name содержит имя аргумента метода, использование которого показано в следующем примере:

```
public class SampleClass {
/// <paramref name="input"/>-это строка ввода.
public static void SampleMethod (String input) { }
```

<permission>

Тэг <permission> применяется для комментирования прав доступа к элементам вашего кода. Для этого параметр тэга cref=«...» должен содержать System.Security.PermissionSet. В следующем примере демонстрируется использование этого тэга:

```
using System;
class SampleClass{
/// <exception cref="System.Security.PermissionSet"> Свободный доступ к этому
методу </exception>
public static void SampleMethod() { }
public static void Main() { }
```

Результат обработки этого примера показан ниже:

```
<?xml version="1.0"?>
<doc>
 <members>
 <member name="M:SampleClass.SampleMethod">
 <permission cref="T:System.Security.PermissionSet"> Свободный доступ к этому
методу </permission>
 </member>
 </members>
</doc>
```

<remarks>

Тэг <remarks> служит для вставки комментариев для вашего класса или другого типа.

<returns>

Тэг <returns> используется для описания возвращаемого значения метода. Следующий пример демонстрирует его применение:

```
using System;
class SampleClass{
```

```
/// <returns> Возвращает число Пи </returns>
public static double PiSample() { return 3.1415926; }
public static void Main() { }
```

<see>

Тэг <see> позволяет вам ссылаться на любой элемент, доступный в процессе компиляции, например элемент вашего кода. Его параметр cref =«...» — ссылка на этот элемент. Пример:

```
using System;
class SampleClass{
/// <para> Метод Main использует System.Console. <see cref="System.Console"/>
for console operations. </para>
public static void Main() {
System.Console.WriteLine("SampleClass Console Output");
}
```

<seealso>

Тэг <seealso> ссылается на элементы, которые следует поместить в раздел «Смотри также». Параметр этого тэга cref=«...» — ссылка на любой элемент, доступный в процессе компиляции.

<summary>

Тэг <summary> следует использовать для описания элемента вашего класса. Для описания самого класса необходимо выбрать <remarks>. Пример:

```
using System;
/// <remarks> Это мой класс </remarks>
class SampleClass{
/// <summary> Main это метод класса SampleClass </summary>
public static void Main() { )
```

<value>

Тэг <value> используется для описания свойств.

ПРАВИЛА ДОКУМЕНТИРОВАНИЯ

Как вы уже могли заметить, XML комментарий кода начинается с ///. Помимо того, компилятор проверяет синтаксис всех тэгов и допустимость употребления большинства тэгов. Например, если вы захотите описать несуществующий аргумент метода, компилятор выдаст ошибку.

Вам могло показаться, что некоторые тэги являются бесполезными, без них можно обойтись. Необходимо отметить, что XML не несет в себе представления, а правильная структура ваших XML комментариев гарантирует правильное их представление. Именно поэтому я советую вам воспользоваться приведенными выше рекомендациями.

18. РАБОТА СО СТРОКАМИ

Было время, когда люди использовали компьютеры исключительно для управления числовыми значениями. Первые компьютеры нужны были лишь для вычисления траекторий полета ракет, и программирование преподавалось только на математических факультетах ведущих университетов.

Сегодня большинство программ оперирует со строками намного больше, чем с числами. Обычно строки используются для обработки текстов, манипуляций с документами и создания Web-страниц.

C# обеспечивает встроенную поддержку работы со строками. Более того, C# обрабатывает строки как объекты, что инкапсулирует все методы манипуляции, сортировки и поиска, обычно применяемые к строкам символов.

ОСОБЕННОСТИ ТИПА Sytem.String

C# обрабатывает строки как встроенные типы, которые являются гибкими, мощными и удобными. Каждый объект строки — это неизменная последовательность символов Unicode. Другими словами, те методы, которые изменяют строки, на самом деле возвращают измененную копию, а первоначальная строк остается неповрежденной.

Объявляя строку на C# с помощью ключевого слова string, вы фактически объявляете объект типа System.String, являющийся одним из встроенных типов, обеспечиваемых .NET Framework библиотекой классов. C# строка — это объект типа System.String, и я буду употреблять эти имена попеременно на протяжении всей главы.

Объявление класса System.String следующее:

```
public sealed class String: IComparable, ICloneable, IConvertible, IEnumerable
```

Такое объявление говорит о том, что класс запечатан, что невозможно унаследовать свой класс от класса String. Класс также реализует четыре системных интерфейса — IComparable, ICloneable, IConvertible и IEnumerable — которые определяют функциональные возможности System.String за счет его дополнительного использовния с другими классами в .NET Framework.

Интерфейс IComparable определяет тип, реализующий его как тип, значения которого могут упорядочиваться. Строки, например, могут быть расположены в алфавитном порядке; любую строку можно сравнить с другими строками, чтобы определить, какая из них должна стоять пер-

вой в упорядоченном списке. IComparable классы реализуют метод CompareTo.

IEnumerable интерфейс позволяет вам использовать инструкцию foreach, чтобы перебирать элементы строки как набор символов.

ICloneable объекты могут создавать новые экземпляры объектов с теми же самыми значениями, как и первоначальный вариант. В данном случае возможно клонировать строку таким образом, чтобы создать новую с тем же самым набором символов, как и в оригинале. ICloneable классы реализуют метод Clone().

IConvertible классы реализуют методы для облегчения преобразования объектов класса к другим встроенным типам, например ToInt32 (), ToDouble (), ToDecimal () и т. д.

СОЗДАНИЕ СТРОК

Наиболее общий способ создания строк состоит в том, чтобы установить строку символов, известную как строковый литерал, определяемый пользователем, в переменную типа string:

```
string newString = " Новая строка";
```

Указанная строка может содержать служебные символы типа «\n» или «\t», которые начинаются с наклонной черты (\) и используются для указания перевода строки или вставки символа табуляции. Поскольку наклонная черта влево самостоятельно используется в некоторых синтаксисах строк, типа URL или путей каталога, то в такой строке наклонной черте влево должен предшествовать другой символ наклонной черты влево.

Строки могут также быть созданы с помощью дословной записи строки. Такие строки должны начинаться с символа (@), который сообщает конструктору String, что строка должна использоваться дословно, даже если она включает служебные символы. В дословном определении строки наклонные черты влево и символы, которые следуют за ними, просто рассматриваются как дополнительные символы строки. Таким образом, следующие два определения эквивалентны:

```
string stringOne = "\\\\MySystem\\MyDirectory\\MyFile.txt";
string stringTwo = @"\\MySystem\MyDirectory\MyFile.txt";
```

В первой строке используется обычный литерал строки, так что символы наклонной черты влево (\) должны дублироваться. Это означает, что для отображения (\) нужно записать (\\). Во второй строке используется дословная литеральная строка, так что дополнительная наклонная черта влево не нужна.

Следующий пример иллюстрирует многострочные строки:

```
string stringOne = "Line One\nLine Two";
string stringTwo = @"Line One
Line Two";
```

И снова эти объявления строк взаимозаменяемы. Какой вариант вы будете использовать — вопрос удобства и личного стиля.

System.Object.ToString()

Другой способ создать строку состоит в том, чтобы вызвать у объекта метод ToString() и установить результат переменной типа string. Все встроенные типы имеют этот метод, что позволяет упростить задачу преобразования значения (часто числового значения) к строковому виду. В следующем примере вызывается метод ToString() для типа int, чтобы сохранить его значение в строку.

```
int myInt = 10;
string intString = myInt.ToString( ) ;
```

Вызов метода ToString() у объекта mylnt вернет строковое представление числа 10.

Класс System.String в .NET поддерживает множество перегруженных конструкторов, которые обеспечивают разнообразные методы для инициализации строковых значений различными типами. Некоторые из этих конструкторов дают возможность создавать строку в виде массива символов или в виде указателя на символы. При создании строки в виде массива символов CLR создает экземпляр новой строки с использованием безопасного кода. При создании строки на основе указателя применяется «небезопасный» код, что крайне нежелательно при разработке приложений .NET.

МАНИПУЛИРОВАНИЕ СТРОКАМИ

Класс string обеспечивает множество встроенных методов для сравнения, поиска и управления строковыми значениями. Вот неполный список всех возможностей этого класса:

Empty — свойство, определяющее, пустая ли строка;
Compare() — функция сравнения двух строк;
CompareOrdinal() — сравнивает строки в независимости от региональных настроек;
Concat() — создает новую строку из двух и более исходных строк;
Copy() — создает дубликат исходной строки;
Equals() — определяет, содержат ли две строки одинаковые значения;
Format( ) — форматирует строку, используя строго заданный формат;
InternO — возвращает ссылку на существующий экземпляр строки;
Join() — добавляет новую строку в любое место уже существующей строки;
Chars — индексатор символов строки;
Length — количество символов в строке;
Clone() — возвращает ссылку на существующую строку;
CompareTo() — сравнивает одну строку с другой;
CopyToQ — копирует определенное число символов строки в массив Unicode символов;
EndsWith() — определяет, заканчивается ли строка определенной последовательностью символов;

Equals() — определяет, имеют ли две строки одинаковые значения;
Insert() — вставляет новую строку в уже существующую;
LastIndexOf() — возвращает индекс последнего вхождения элемента в строку;
PadLeft() — выравнивает строку по правому краю, пропуская все пробельные символы или другие (специально заданные);
PadRight() — выравнивает строку по левому краю, пропуская все пробельные символы или другие (специально заданные);
Remove() — удаляет необходимое число символов из строки;
Split() — возвращает подстроку, отделенную от основного массива определенным символом;
StartsWith() — определяет, начинается ли строка с определенной последовательности символов;
Substring() — возвращает подстроку из общего массива символов;
ToCharArray() — копирует символы из строки в массив символов;
ToLower() — преобразует строку к нижнему регистру;
ToUpper() — преобразует строку к верхнему регистру;
Trim() — удаляет все вхождения определенных символов в начале и в конце строки;
TrimEnd() — удаляет все вхождения определенных символов в конце строки;
TrimStart() — удаляет все вхождения определенных символов в начале строки.

Теперь давайте рассмотрим пример использования строк. Для этого напишем приложение, использующее методы Compare( ), Concat( ), Copy(), Insert( ) и многие другие.

```
using System;
using System;

namespace StringManipulating
{
  using System;
  public class StringTester
  {
    static void Main( )
    {
      // создаем несколько строк
      string str1 = "абвг";
      string str2 = "АБВГ";
      string str3 = @"C# представляет собой инструмент " +
          "быстрого создания приложений для .NET платформы";
      int result;

      //методы сравнения строк
```

```
//используем статическую функцию Compare для сравнения
result = string.Compare(str1, str2);
Console.WriteLine( @"сравниваем str1; {0} и str2: {1}, "+
  "результат; {2}\n", str1, str2, result);

//используем функцию Compare с дополнительным параметром
//для игнорирования регистра строки
result = string.Compare(str1, str2, true);
Console.WriteLine("Сравниваем без учета регистра");
Console.WriteLine(@"str1: {0), str2: {1}, "+
    "результат: {2}\n", str1, str2, result);

// методы объединения строк

//используем функцию Concat для объединения строк
string str4 = string.Concat(str1, str2);
Console.WriteLine( @"Создаем str4 путем "+
  "объединения str1 и str2: \n{0}", str4);

// используем перегруженный оператор
// для объединения строк
string str5 = str1 + str2;
Console.WriteLine( "\nstr5 = str1 + str2 =:   {0}\n", str5);

// используем метод Copy для копирования строки
string str6 = string.Copy(str5);
Console.WriteLine( "\nstr6 скопирована из str5: {0}\n", str6);

// используем перегруженный оператор копирования
string str7 = str6;
Console.WriteLine("str7 = str6: {0}", str7);

// несколько способов сравнения

// используя метод Equals самого объекта
Console.WriteLine f "\nstr7.Equals(str6)?: {0}",
  str7.Equals(str6));
//используя статический метод Equals
Console.WriteLine( "\nstr7 и str6 равны?; {01",
  string.Equals(str7, str6));
//используя оператор ==
Console.WriteLine{ "\nstr7 == str6 ?; {0j",
  str7 == str6);

//определение длины строки
```

```
    Console.WriteLine( "\nСтрока str7 имеет длину {0} символов. ",
      str7.Length);

    //определение символа строки по его индексу
    Console.WriteLine(
      "Пятым элементом в строке str7 является символ {0}\n",
      str7[4]);

    //сравнение конца строки с входным экземпляром
    Console.WriteLine(
      "str3:{0}\nЗаканчивается ли эта строка словом \"инструмент\"?: {1}\n",
      str3, str3.EndsWith("инструмент") );

    //сравнение конца строки с входным экземпляром
    Console.WriteLine(
      "str3:{0}\nЗаканчивается ли эта строка словом \"платформа\"?: {1}\n",
      str3, str3.EndsWith("платформы") );

    //поиск первого вхождения подстроки в строку
    Console.WriteLine( @"Первое вхождение слова инструмент "+
        "в строку str3 имеет индекс {0}\n", str3.IndexOf("инструмент"));

    // вставляем новое слово в строку
    string str8 = str3.Insert(str3.IndexOf("приложений"),"мощных " ) ;
    Console.WriteLine("str8: {0i\n", str8);
  }
}
```

Результат работы программы будет следующий:

сравниваем str1: абвг и str2: АБВГ, результат: -1

Сравниваем без учета регистра
strl: абвг, str2: АБВГ, результат: 0

Создаем str4 путем объединения strl и str2:
абвгАБВГ

str5 = strl + str2 =: абвгАБВГ

str6 скопирована из str5: абвгАБВГ

str7 = str6: абвгАБВГ

str7.Eguals(str6)?: True

str7 и str6 равны?: True

str7 == str6 ?: True

Строка str7 имеет длину 8 символов.

Пятым элементом в строке str7 является символ А

str3:C# представляет собой инструмент быстрого создания приложений для .NET платформы
Заканчивается ли эта строка словом "инструмент"?: False

str3:C# представляет собой инструмент быстрого создания приложений для .NET платформы
Заканчивается ли эта строка словом "платформа"?: True

Первое вхождение слова инструмент в строку str3 имеет индекс 22

str8: C# представляет собой инструмент быстрого создания мощных приложений для.
NET платформы

Вы можете использовать различные способы объявления строк. Для объявления строки str3 я использовал дословное представление строки. Следует отметить, что вы можете разрывать строку в коде программы для переноса ее на другую линию. При этом необходимо объединять разорванные части строки оператором (+). Такая строка будет восприниматься как единое целое.

```
string str3 = @"C# представляет собой инструмент " +
    "быстрого создания приложений для .NET платформы";
     result = string.Compare(str1, str2);
     Console.WriteLine( @"сравниваем str1: {0} и str2: {1}, "+
       "результат: {2}\n", str1, str2, result);
```

В данном случае используется чувствительная к регистру функция сравнения двух чисел. Функция сравнения всегда возвращает различные значения, в зависимости от результата сравнения:

- значение меньше нуля, если первая строка меньше второй;
- 0, если строки равны;
- значение больше нуля, если первая строка больше второй.

В нашем случае результат будет следующий:

сравниваем str1: абвг и str2: АБВГ, результат: -1

Буквы нижнего регистра имеют меньшее значение, нежели верхнего, отсюда и результат.

В следующей функции Compare мы используем сравнение без учета регистра. Об этом свидетельствует дополнительный третий параметр функции — true.

```
result = string.Compare(str1, str2, true);
Console.WriteLine("Сравниваем без учета регистра");
Console.WriteLine(@"str1: {0}, str2: {1}, " +
    "результат: {2}\n", str1, str2, result);
```

Соответственно и результат будет:

Сравниваем без учета регистра
str1: абвг, str2: АБВГ, результат: О

Функция сравнения без учета регистра сначала приводит обе строки к общему регистру, а затем осуществляет посимвольное сравнивание строк. В итоге мы получаем последовательность действий:

абвг → АБВГ, АБВГ = АБВГ ? → ДА → результат О

Для объединения строк мы использовали две возможности класса string. Одна из них — это использование статической функции Concat().

```
string str4 = string.Concat(str1, str2];
```

Второй способ — использование оператора (+).

```
string str5 = str1 + str2;
```

Оператор (+) класса string перегружен таким образом, что выполняет действие, аналогичное функции Concat(). Однако использование записи str1+str2 лучше читаемо, поэтому программисты обычно предпочитают применение операторов вызову функций.

Аналогичное сравнение можно провести между функцией Copy() и оператором (=). Они выпоняют одно и то же действие — копируют содержимое одной строки в другую. Разница состоит лишь в записи кода программы:

```
string str6 = string.Copy(str5);
string str7 = str6;
```

И str6, и str7 будут в результате иметь то значение, которое записано в строке str5.

Класс string в C# обеспечивает три способа проверки равенства двух строк. Первый из них — это использование метода Equals().

```
str7.Equals(str6)
```

В данном случае для объекта str7 проверяется равенство ему объекта str6. Вторым вариантом проверки равенства строк является использование статической функции string.Equals().

```
string.Equals(str7, srr6)
```

Третий вариант — это использование перегруженного оператора (==).

```
str7 == str6
```

Любой из этих вызовов возвращает True, если строки равны, и False, если строки не равны.

Свойство Length возвращает число символов строки. А оператор ([]) возвращает символ строки, имеющий соответствующий индекс.

```
Console.WriteLine(
"Пятым элементом в строке str7 является символ {0}\n", str7[4]);
```

Здесь мы пытаемся получить пятый элемент строки, используя число 4 в операторе ([]). Все потому, что элементы строки в C#, как и в C++, начинаются с нулевого индекса.

Строка	а	б	в	г	А	Б	В	Г
Индекс элемента	0	1	2	3	4	5	6	7
Порядковый номер элемента	1	2	3	4	5	6	7	**8**

Из этой таблицы видно, что пятым символом строки является символ «А», имеющий индекс 4.

ПОИСК ПОДСТРОКИ

Тип string имеет перегруженный метод Substring() для извлечения подстроки из строки. Один из методов принимает в качестве параметра индекс элемента, начиная с которого следует извлечь подстроку. Второй метод принимает и начальный, и конечный индекс, чтобы указать, где начать и где закончить поиск. Метод Substring можно рассмотреть на следующем примере. Программа выводит слова строки в порядке, обратном последовательности их записи:

```
using System;
using System.Text;

namespace C_Sharp_Examples
{
  public class StringFinder
  {
    static void Main( )
    {
      // объявляем строку для обработки
      string s1 = "Один Два Три Четыре";

      // получаем индекс последнего пробела
      int ix = s1.LastIndexOf(" " ) ;

      // получаем последнее слово в строке
      string s2 = s1.Substring(ix+1);

      // устанавливаем s1 на подстроку начинающуюся
      //с 0-ого индекса и заканчивающуюся последним пробелом
      s1 = s1.Substring(0, ix) ;

      // вновь ищем индекс последнего пробела
      ix = s1.LastIndexOf(" " ] ;

      // устанавливаем s3 на последнее слово строки
      string s3 = s1.Substring(ix+1);
```

```
        // сбрасываем s1 на подстроку
        // от нулевого символа до ix
        s1 = s1.Substring(0,ix);

        // сбрасываем ix на пробел
        // между "один" и "два"
        ix = s1.LastIndexOf(" ");

        // устанавливаем s4 на подстроку после ix
        string s4 = s1.Substring(ix+1);

        //устанавливаем s1 на подстроку от 0 до ix
        //получаем только слово "один"
        s1 = s1.Substring(0,ix);

        // пытаемся получить индекс последнего пробела
        // но на этот раз функция LastIndexOf вернет -1
        ix = s1.LastIndexOf(" " );

        // устанавливаем s5 на подстроку начиная с ix+1
        // поскольку ix = 1, s5 устанавливается на начало s1
        string s5 = s1.Substring(ix+1);

        // Выводим результат
        Console.WriteLine ("s2: {0}\ns3: {1}",s2,s3);
        Console.WriteLine ("s4: {0}\ns5: {1}\n",s4,s5);
        Console.WriteLine ("s1: {0}\n",s1);

    }
  }
}
```

Первым делом объявляем строку и инициализируем ее необходимыми параметрами.

```
string s1 = "Один Два Три Четыре";
```

Затем мы вычисляем позицию последнего пробела в строке. Это необходимо для того, чтобы определить начало последнего слова строки.

```
int ix = s1.LastIndexOf(" ");
```

В данном случае значение ix будет равно 12. Слово «Четыре» начинается с позиции 13. Теперь, когда мы знаем начало последнего слова строки, можно извлечь его.

```
string s2 = s1.Substring(ix+1);
```

В итоге s2 будет равно «Четыре». Далее мы обрезаем исходную строку s1 на слово «Четыре». Для этого необходимо вызвать функцию Substring с двумя параметрами — начала и конца подстроки. Началом строки у нас будет начало исходной строки, а концом — индекс последнего пробела.

```
s1 = s1.Substring(0, ix);
```

Новая строка будет иметь вид «Один Два Три». Теперь мы повторяем ту же последовательность действий, что и ранее для полной строки. Получаем индекс последнего пробела, выбираем последнее слово, обрезаем строку ... Делаем это до тех пор, пока в строке не останется одно слово «Один». Когда мы попытаемся получить из этой строки индекс символа пробела, то функция вернет значение -1.

```
ix = s1.LastIndexOf(" ");
```

Поэтому, при вызове функции Substring() и передаче в нее значения (-1 + 1 = 0), нам вернется полная исходная строка, а именно слово «Один».

```
string s5 = s1.Substring(ix+1);
```

Результат работы программы будет следующий:

s2: Четыре
S3: Три
s4: Два
s5: Один

s1: Один

РАЗБИЕНИЕ СТРОК

По сути, предыдущий пример делал разбор строки на слова, сохранял найденные слова и выводил их на экран. Более эффективное решение проблемы, проиллюстрированной в предыдущем примере, состоит в том, чтобы использовать метод Split() класса string. Метод Split() разбирает строку в подстроки. Для вьщеления подстрок из исходной строки необходимо передать методу Split() разделительный символ в качестве параметра. При этом результат вернется в виде массива строк. Давайте рассмотрим работу метода Split() на примере:

```
using System;
using System.Text;

namespace C_Sharp_Examples
{
  public class StringTester
  {
    static void Main( )
    {
      // строка для анализа
      string s1 = "Один,Два,Три, Строка для разбора";

      // задаем разделительные символы для анализа
      const char cSpace = ' ';
      const char cComma = ',';

      // создаем массив разделительных символов
```

```
        // и инициализируем его двумя элементами
        char[] delimiters = new char[] { cSpace, cComma };

        string output = "";
        int ctr = 1;

        // выделяем подстроки на основе разделителей
        // и сохраняем результат
        foreach (string substring in s1.Split(delimiters))
        {
          output += ctr++; //номер подстроки
          output += ": ";
          output += substring; //подстрока
          output += "\n";       //переход на новую линию
        }

        // вывод результата
        Console.WriteLine(output);
      }
    }
}
```

Результатом работы программы будет текст:

1: Один
2: Два
3: Три
4:
5: Строка
6: для
7: разбора

Обратите внимание, что строка под номером 4 пуста — это не ошибка. Ее особенность мы сейчас рассмотрим.

```
string s1 = "Один,Два,Три, Строка для разбора";
```

Мы объявили строку **s1**, содержащую шесть слов. В строке используется два типа разделительных символов. Один из них символ пробела, другой — символ (,). Между словами «Три» и «Строка» располагаются сразу два разделительных символа, запятая и пробел. Это очень важно отметить!

Далее в программе объявляются две константы, определяющие разделительные символы.

```
const char cSpace = ' ';
const char cComma = ',';
```

Эти константы объединяются в один массив.

```
char[] delimiters = new char[] { cSpace, cComma };
```

Метод Split() может принимать в качестве параметра как один символ, так и массив разделительных символов. В нашем случае мы используем в качестве параметра массив разделительных символов из двух элемен-

тов — символа пробела и символа запятой. Следовательно, как только метод Split() обнаружит в исходной строке один из этих символов, то сразу же поместит в выходной массив последовательность символов от предыдущего разделителя до текущей позиции.

```
foreach (string substring in s1.Split(delimiters))
```

Инструкция foreach в данном случае будет применяться ко всем элементам результирующего массива строк, который вернет функция Split(). Результирующий массив будет состоять из семи элементов, одним из которых окажется пустая строка. Пустая строка появилась из-за характерной особенности анализируемой строки. Как я уже отмечал ранее, между словами «Три» и «Строка» располагаются сразу два разделительных символа — и запятая, и пробел. Когда Метод Split() дойдет до анализа символа пробела между «Три» и «Строка», он определит пробел как разделительный символ. Однако сразу перед ним находится еще один разделительный символ. Следовательно, между двумя разделительными символами располагается строка длиной в 0 символов — пустая строка, которую метод Split() и помещает в выходной массив.

КЛАСС StringBuilder

Класс StringBuilder используется для создания и редактирования строк, обычно строк из динамического набора данных, например из массива байтовых значений. Наиболее важными членами класса StringBuilder являются:

Capacity — определяет число символов, которые способен хранить и обрабатывать StringBuilder;
Chars — индексатор класса;
Length — определяет длину объекта StringBuilder;
MaxCapacity — определяет максимальное число символов, которые способен хранить и обрабатывать StringBuilder;
Append() — добавляет объект заданного типа в конец StringBuilder;
AppendFormat() — замещает или устанавливает новый формат StringBuilder;
EnsureCapacity() — гарантирует, что StringBuilder имеет емкость не менее указанной в параметре;
Insert() — вставляет объект некоторого типа в указанную позицию;
Remove() — удаляет указанный символ;
Replace() — замещает все экземпляры указанных символов на новые символы.

Очень важной особенностью класса StringBuilder является то, что при изменении значений в объекте StringBuilder происходит изменение значений в исходной строке, а не в ее копии. Давайте рассмотрим пример использования класса StringBuilder для работы со строками. Возьмем для вывода результатов в предыдущем примере вместо класса string класс StringBuilder.

```
using System;
using System.Text;

namespace C_Sharp_Examples
{
  public class StringBuilderOutput
  {
    static void Main( )
    {
      // строка для анализа
      string s1 = "Один,Два,Три, Строка для разбора";

      // задаем разделительные символы для анализа
      const char cSpace = ' ';
      const char cComma = ',';

      // создаем массив разделительных символов
      // и инициализируем его двумя элементами
      char[] delimiters = new char[] { cSpace, cComma };

      StringBuilder output = new StringBuilder();
      int ctr = 1;

      // выделяем подстроки на основе разделителей
      // и сохраняем результат
      foreach (string substring in s1.Split(delimiters))
      {
        // AppendFormat добавляет строку определенного формата
        output.AppendFormat("{0}: {1}\n", ctr++, substring);
      }

      // вывод результата
      Console.WriteLine(output);
    }
  }
}
```

Как вы могли заметить, изменения произошли лишь в нижней части кода программы. Работа по разбору строки происходит как обычно, изменился лишь вывод результатов. Для вывода результатов мы используем класс StringBuilder.

```
StringBuilder output = new StringBuilder();
```

Для объектов класса StringBuilder всегда нужно явно вызывать конструктор, поскольку этот тип не относится к встроенным типам. При сохранении результатов обработки строки используется метод AppendFormat(). Данный метод позволяет отформатировать строку необходимым форматом. Мы передаем в качестве формата строки два значения, разделенные

символом двоеточия. Первое значение — это номер подстроки, второе — сама подстрока.

```
output.AppendFormat("{0} : {1}\n", ctr++, substring);
```

При выводе результата нам не потребуется проводить дополнительное форматирование текста, поскольку мы сразу заносили результат в нужном формате. Мы просто выводим содержимое StringBuilder в консоль.

```
Console.WriteLine(output);
```

Результат работы программы остается таким же, что и ранее:

1: Один
2: Два
3: Три
4:
5: Строка
6: для
7: разбора

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Регулярные выражения — это один из способов поиска подстрок (соответствий) в строках. Осуществляется с помощью просмотра строки в поисках некоторого шаблона. Общеизвестным примером могут быть символы «*» и «?», используемые в командной строке DOS. Первый из них заменяет ноль или более произвольных символов, второй же — один произвольный символ. Так, использование шаблона поиска типа «text?.*» найдет файлы textf.txt, textl.asp и другие аналогичные, но не найдет text.txt или text.htm. Если в DOS использование регулярных выражений было крайне ограничено, то в других местах (то есть операционных системах и языках программирования) они почти достигли уровня высокого искусства.

Применение регулярных выражений дает значительное увеличение производительности, поскольку библиотеки, интерпретирующие регулярные выражения, обычно пишутся на низкоуровневых высокопроизводительных языках (C, C++, Assembler).

Применение регулярных выражений

Обычно с помощью регулярных выражений выполняются три действия:

- проверка наличия соответствующей шаблону подстроки;
- поиск и вьщача пользователю соответствующих шаблону подстрок;
- замена соответствующих шаблону подстрок.

В C# работа с регулярными выражениями выглядит следующим образом:

```
Regex re = new Regex("образец", "опции");
MatchCollection mc = re.Matches("строка для поиска");
iCountMatchs = mc.Count;
```

re — это объект типа Regex. В конструкторе ему передается образец поиска и опции. Ниже представлена таблица, задающая различные варианты поиска.

Символ	Значение
I	Поиск без учета регистра.
m	Многострочный режим, позволяющий находить совпадения в начале или конце строки, а не всего текста.
n	Находит только явно именованные или нумерованные группы в форме (?<name>...).(Значение этого будет объяснено ниже, при обсуждении роли скобок в регулярных выражениях.)
c	Компилирует. Генерирует промежуточный MSIL-код, перед исполнением превращающийся в машинный код.
s	Позволяет интерпретировать конец строки как обыкновенный символ-разделитель. Часто это значительно упрощает жизнь.
x	Исключает из образца неприкрытые незначащие символы (пробелы, табуляция и т.д.).
r	Ищет справа налево.

Сочетание флагов m и s дает очень удобный режим работы, учитывающий концы строк и позволяющий пропустить все незначащие символы, включая символ конца строки.

Основы синтаксиса регулярных выражений

Не стану пытаться написать полный справочник по всем символам, используемым в шаблонах регулярных выражений, а приведу только основные метасимволы.

Выражением может быть один символ или последовательность символов, заключенных в круглые или квадратные скобки. Особенности использования скобок будут описаны ниже.

Классы символов (Character classes)

Используя квадратные скобки, можно указать группу символов (это называют классом символов) для поиска. Например, конструкция «б[а-и]ржа» соответствует словам «баржа» и «биржа», т. е. словам, начинающимся с «б», за которым следуют «а» или «и», и заканчивающимся на «ржа».

Возможно и обратное, то есть можно указать символы, которые не должны содержаться в найденной подстроке. Так, '[^1-6]' находит все символы, кроме цифр от 1 до 6. Следует упомянуть, что внутри класса символов '\b' обозначает символ backspace (стирания).

Квантификаторы, или умножители (Quantifiers)

Если неизвестно, сколько именно знаков должна содержать искомая подстрока, можно использовать спецсимволы, именуемые мудреным словом *квантификаторы* (quantifiers). Например, можно написать «hel+o», что будет означать слово, начинающееся с «he», со следующими за ним одним или несколькими «l», и заканчивающееся на «o». Следует понимать, что квантификатор относится к предшествующему выражению, а не к отдельному символу. Полный список квантификаторов можно увидеть в нижеприведенной таблице.

Символ	Описание
*	Соответствует 0 или более вхождений предшествующего выражения. Например, 'zo*' соответствует "z" и "zoo".
+	Соответствует 1 или более предшествующих выражений. Например, "zo+" соответствует "zo" и "zoo", но не "z".
?	Соответствует 0 или 1 предшествующих выражений. Например, 'do(es)?' соответствует "do" в "do" или "does".
{n}	n — неотрицательное целое. Соответствует точному количеству вхождений. Например, 'o{2}' не найдет "o" в "Bob", но найдет два "o" в "food".
{n,}	n — неотрицательное целое. Соответствует вхождению, повторенному не менее п раз. Например, 'o{2,}' не находит "o" в "Bob", зато находит все "o" в "foooood". 'o{1,}' эквивалентно 'o+'. 'o{0,}' эквивалентно 'o*'.
{n,m}	m и n— неотрицательные целые числа, где п <= т. Соответствует минимум п и максимум m вхождений. Например, 'o{1,3} находит три первые "o" в "fooooood". 'o{0,1}' эквивалентно 'o?'. Пробел между запятой и цифрами недопустим.

Концы и начала строк

Проверка начала или конца строки производится с помощью метасимволов ^ и $. Например, «^thing» соответствует строке, начинающейся с «thing». «thing$» соответствует строке, заканчивающейся на «thing». Эти символы работают только при включенной опции 's'. При выключенной опции 's' находятся только конец и начало текста. Но и в этом случае можно найти конец и начало строки, используя escape-последовательности \A и \Z.

Граница слова

Для задания границ слова используются метасимволы '\Ъ' и '\B'.

```
Regex re = new Regex("меня", "ms");
```

В данном случае образец в re соответствует не только «меня» в строке «найди меня», но и «меня» в строке «родители поменяли квартиру». Чтобы избежать этого, можно предварить образец маркером границы слова:

```
Regex re = new Regex("\bменя", "ms");
```

Теперь будет найдено только «меня» в начале слова. Не стоит забывать, что ВНУТРИ класса символов '\b' обозначает символ backspace (стирания).

Приведенные ниже в таблице метасимволы не заставляют машину регулярных выражений продвигаться по строке или захватывать символы. Они просто соответствуют определенному месту строки. Например, ^ определяет, что текущая позиция — начало строки. '^FTP' возвращает только те «FTP», что находятся в начале строки.

Символ	Значение
^	Начало строки.
$	Конец строки, или перед \n в конце строки (см. опцию m).
\A	Начало строки (ignores the m option).
\Z	Конец строки, или перед \n в конце строки (игнорирует опцию т).
\z	Точно конец строки (игнорирует опцию т).
\G	Начало текущего поиска (часто это в одном символе за концом последнего поиска).
\b	На границе между \w (алфавитно-цифровыми) и \W (неалфавитно-цифровыми) символами. Возвращает true на первых и последних символах слов, разделенных пробелами.
\B	Не на \b-границе.
\w	Слово. То же, что и [a-zA-Z_0-9].
\W	Все, кроме слов. То же, что и [^a-zA-Z_0-9].
\s	Любое пустое место. То же, что и [\f\n\r\t\v].
\S	Любое непустое место. То же, что и [^ \f\n\r\t\v].
\й	Десятичная цифра. То же, что и [0-9].
YD	Не цифра. То же, что и [^0-9].

Вариации и группировка

Символ 'I' можно использовать для перебора нескольких вариантов. Употребление его вместе со скобками — '(... | ... |...)' — позволяет создать группы вариантов. Скобки используются для «захвата» подстрок для дальнейшего использования и сохранения их во встроенных переменных $1, $2,..., $9.

```
Regex re = new Regex("like (apples|pines|bananas)");
MatchCollection mc = re.Matches("I like apples a lot");
```

Такой пример будет работать и найдет последовательность «like apples», поскольку «apples» — один из трех перечисленных вариантов. Скобки также поместят «apples» в $1 как обратную ссылку для дальнейшего использования. В основном это имеет смысл при замене.

ИСПОЛЬЗОВАНИЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ: Regex

Я думаю, с вас достаточно теории. Давайте перейдем к практике. .Net технология обеспечивает объектно-ориентированный подход к обработке регулярных выражений.

Библиотека классов для обработки регулярных выражений основывается на пространстве имен System.Text.RegularExpressions. Главным классом для обработки регулярных выражений является класс Regex, который представляет собой компилятор регулярных выражений. Кроме того, Regex обеспечивает множество полезных статических методов. Использование Regex проиллюстрировано в следующем примере:

```
using System;
using System.Text;
using System.Text.RegularExpressions;

namespace Regular_Expressions
{
  public class Tester
  {
    static void Main( )
    {
      string s1 = "Один,Два,Три, Строка для разбора";
      Regex theRegex = new Regex(" |, |,");
      StringBuilder sBuilder = new StringBuilder( );
      int id = 1;
      foreach (string substring in theRegex.Split(s1))
      {
        sBuilder.AppendFormat( "{0}: {1}\n", id++, substring);
      }
      Console.WriteLine("{0}", sBuilder);
    }
  }
}
```

Результат работы программы будет представлен как:

1: Один
2: Два
3: Три
4: Строка
5: для
6: разбора

Как вы могли заметить, в этом примере есть та же строка, что и в примере для поиска подстроки «Один, Два, Три, Строка для разбора». Мы объявили ее как s1. Далее мы объявили регулярное выражение, которое задает образец-разделитель для поиска:

```
Regex theRegex = new Regex(" |, |,");
```

В качестве образца используется выражение, объединенное оператором ИЛИ. Ищется либо знак пробела, либо запятая, либо идущие подряд запятая и пробел. Строка, задающая регулярное выражение, передается в качеств параметра конструктору объекта theRegex.

Класс Regex содержит метод Split, который по своему действию напоминает метод string.Split. Он возвращает массив строк как результат поиска регулярного выражения в строке. В теле цикла уже привычным для вас способом, при помощи класса String Builder, формируется выходная строка.

Метод Split класса Regex перегружен и может использоваться в двух вариантах. Первый из них был показан в предыдущем примере. Второй способ — использование вызова статического метода.

```
using System;
using System.Text;
using System.Text.RegularExpressions;

namespace Regular_Expressions
{
  public class Tester
  {
    static void Main( )
    {
      string s1 = "Один,Два,Три, Строка для разбора";
      StringBuilder sBuilder = new StringBuilder( );
      int id = 1;
      foreach (string substring in Regex.Split(s1, " |, |,"))
      {
        sBuilder.AppendFormat( "{0}: {1}\n", id++, substring);
      }
      Console.WriteLine("{0}", sBuilder);
    }
  }
}
```

Результатом работы программы станет все тот же выходной массив строк. Отличие этого примера от предыдущего лишь в том, что нам не пришлось создавать экземпляр объекта Regex. В данном случае для поиска использовался статический метод Regex.Split, который принимает два параметра — строку, в которой будет производиться поиск, и строку с регулярным выражением.

Кроме этого, метод Split перегружен еще двумя вариантами, которые позволяют ограничить количество раз использования метода Split и задать начальную позицию в строке для поиска.

Использование Match коллекций

Пространство имен RegularExpressions содержит два класса, которые позволяют осуществлять итерационный поиск в строке и возвращать результат в виде коллекции. Коллекция возвращается в виде объекта типа MatchCollection, содержащего объекты типа Match. Объект Match заключает в себе два очень важных свойства, которые определяют его длину (Length) и значение (Value). Давайте рассмотрим пример использования итерационного поиска:

```
using System;
using System.Text;
using System.Text.RegularExpressions;

namespace Regular_Expressions
{
  class Test
  {
    public static void Main( )
    {
      string s1 = "Это строка для поиска";
      // найти любой пробельный символ
      // следующий за непробельным
      Regex theReg = new Regex(@"(\S+)\s");
      // получить коллекцию результата поиска
      MatchCollection theMarches =
        theReg.Matches(s1);

      // перебор всей коллекции
      foreach (Match theMatch in theMatches)
      {
        Console.WriteLine( "theMatch.Length: {0}",
          theMatch.Length);
        if (theMatch.Length != 0)
        {
          Console.WriteLine("theMatch: {0}",
            theMatch.ToString());
        }
      }
    }
  }
}
```

Результат работы программы:

theMatch.Length: 4
theMatch: Это
theMatch.Length: 7

theMatch: строка
theMatch.Length: 4
theMatch: для

Давайте рассмотрим программу подробнее. Создаем простую строку для поиска:

```
string s1 = "Это строка для поиска";
```

Затем формируем регулярное выражение:

```
Regex theReg = new Regex(@"(\S+)\s");
```

Это пример простейшего регулярного выражения. (\S) означает любой непробельный символ. Знак (+), стоящий после (\S), означает, что может быть любое количество непробельных символов. (\s) в нижнем регистре — пробельный символ. В целом, это выражение означает: «Найти все наборы, которые начинаются с непробельного символа и заканчиваются пробельным».

Результат программы отображает лишь три первых слова исходной строки. Четвертое слово не вошло в результирующую коллекцию, потому что после него нет пробельного символа. Если вы добавите пробел в конце предложения, то слово «поиска» также будет найдено и выведено в качестве результата.

Использование групп

Класс Regex позволяет объединять регулярные выражения в группы. Это бывает очень полезно, если необходимо выбрать из исходной строки все подстроки, удовлетворяющие определенной категории, и таких категорий несколько. Например, нужно выбрать все подстроки, которые могут являться IP адресами и объединить их вместе. Для этого существует класс Group. Любое регулярное выражение может быть добавлено в группу и представлять свою группу по имени. Чтобы не запутаться, давайте перейдем к примеру.

Для начала необходимо уяснить, как создавать группу. Например, для того чтобы создать группу, определяющую IP адрес, вы можете создать следующее регулярное выражение:

```
@"(?<ip>(\d|\.)+)\s"
```

Имя группы задается в о скобках, перед которыми ставится знак (?). Далее следует описание регулярного выражения, определяющего группу.

Класс Match является производным от класса Group и содержит коллекцию объектов Group, к которым можно обратиться через свойство Groups.

```
using System;
using System.Text.RegularExpressions;

namespace Regular_Exprassions
{
  class Test
  {
    public static void Main( )
```

```
    {
      string string1 = "04:03:27 127.0.0.0 GotDotNet.ru";
      // группа time = одна и более цифр или двоеточий
      // за которым следует пробельный символ
      Regex theReg = new Regex(@"(?<время>(\d|\:)+)\s" +
        // группа ip адрес - одна и более цифр или точек
        // за которым следует пробельный символ
        @"(?<ip>(\d|\.)+)\s"
        // группа url = один и более непробельных символов
        @"(?<url>\S+)");

      // получаем коллекцию поиска
      MatchCollection theMatches =
        theReg.Matches(string1);

      // перебор всей коллекции
      foreach (Match theMatch in theMatches)
      {
        if (theMatch.Length != 0)
        {
          // выводим найденную подстроку
          Console.WriteLine("\ntheMatch: {0}",
            theMatch.ToString());

          // выводим группу "time"
          Console.WriteLine("время: {0}",
            theMatch.Groups["время"]);

          // выводим группу "ip"
          Console.WriteLine("ip: {0}",
            theMatch.Groups["ip"]);

          // выводим группу "url"
          Console.WriteLine("url: {0}",
            theMatch.Groups["url"]);
        }
      }
    }
  }
}
```

Результат работы программы:

theMatch: 04:03:27 127.0.0.0 GotDotNet.ru
время: 04:03:27
ip: 127.0.0.0
url: GotDotNet.ru

Вначале мы создали строку для поиска:

```
String string1 = "04:03:27 127.0.0.0 GotDotNet.ru";
```

Эта строка очень простая. Она содержит по одному экземпляру элементов каждой группы: «время», «ip» и «url». Этим группам соответствуют подстроки «04:03:27», «127.0.0.0», «GotDotNet.ru». Но вместо этой строки может быть использована любая другая, например текстовый файл или запрос из базы данных.

Далее в программе создаются группы:

```
Regex theReg = new Regex(
@"(?<время>(\d|\:)+)\s" +
@"(?<ip>(\d|\.)+)\s" +
@"(?<url>\S+)");
```

Здесь создаются три группы: «время», «ip» и «url». Заметьте одну очень важную особенность: можно создавать группы и называть их именами, составленными из букв русского алфавита. Давайте рассмотрим более подробно, каким образом создаются группы.

```
(?<время>
```

«время» — это название группы. Все, что следует после названия группы и до закрывающей скобки, является регулярным выражением, описывающим группу.

```
(?<время>(\d|\:)+)
```

В нашем случае группы разделяются пробельными символами. Точно так же задается группа ip и группа url. Возможно, наши правила описания групп не являются исключительными, но они подойдут для тестового примера.

```
MatchCollection theMatches = theReg.Matches(string1);
```

Здесь происходит поиск совпадений описанных регулярным выражением в строке string1.

Если длина объекта Match превышает значение 0, значит, соответствие найдено. И результат выводится на экран:

```
Console.WriteLine("\ntheMatch: {0}",theMatch.ToString( ) ) ;
```

Далее выводится значение группы «время».

```
Console.WriteLine("время: {0}", theMatch.Groups["время"]);
```

Для этого при помощи строкового индексатора выбирается из GroupCollection группа «время». Затем выводятся значения групп «ip» и «url». Я уже приводил результат работы этой программы. А вот что будет, если строку string1 проинициализировать по-другому. Например, так:

```
string string1 =
  "04:03:27 127.0.0.0 GotDotNec.ru " +
  "04:03:28 127.0.0.0 RSDN.ru " +
  "04:03:29 127.0.0.0 Yandex.ru";
```

В этом случае на экран будет выведен следующий результат:

theMatch: 04:03:27 127.0.0.0 GotDotNet.ru
время: 04:03:27
ip: 127.0.0.0
url: GotDotNet.ru

theMatch: 04:03:28 127.0.0.0 RSDN.ru

время: 04:03:28
ip: 127.0.0.0
url: RSDN.ru

theMatch: 04:03:29 127.0.0.0 Yandex.ru
время: 04:03:29
ip: 127.0.0.0
url: Yandex.ru

В этом случае MatchCollection содержит три объекта, каждый из которых будет выведен на экран. Вы имеете возможность выбирать, что выводить на экран. Это может быть либо полное соответствие выражения, либо определенная группа.

Использование CaptureCollection

Каждый раз, когда объект Regex находит подвыражение, создается экземпляр объекта Capture и добавляется в CaptureCollection. Каждый Cupture объект представляет собой результат единичного поиска. Каждая группа имеет свою CaptureCollection, содержащую подвыражение, соответствующее группе.

Ключевым свойством объекта Capture является его длина, или свойство Length, которое представляет собой длину захваченной строки. Обращаясь к свойству Match.Length, на самом деле вы обращаетесь к свойсту Length объекта Capture, потому что имеет место наследование Match от Group, а Group от Capture.

Объектная модель классов, работающих с регулярными выражениями, позволяет объектам класса Match обращаться к свойствам и методам дочерних классов. В этом смысле объект Group может представлять собой Capture как объединение объектов Capture в единое целое. Объект класса Match инкапсулирует в себе все группы соответствующего поиска по регулярному выражению.

Обычно вам придется иметь дело лишь с одним объектом Capture в CaptureCollection. Но не всегда. Иногда бывает полезно обрабатывать все элементы CaptureCollection. Например, что будет, если в группу попадет несколько экземпляров некоторой подстроки. Давайте изменим предыдущий пример, добавив в строку регулярного выражения группу company. И поместим два экземпляра такой группы в строку с регулярным выражением:

```
Regex theReg = new Regex(
 @"(?<время>(\d|\:)+)\s" +
 @"(?<company>\S+)\s" +
 @"(?<ip>(\d|\.)+)\s" +
 @" (?<company>\S+)\s");
```

Такое регулярное выражение захватывает любое соответствие, начинающееся с группы «время», затем может следовать произвольный набор литеральных символов, затем группа «ip», затем опять набор литераль-

ных символов, заканчивающихся пробелом. Например, разбор следующей строки даст положительный результат:

theMatch: 04:03:27 Double 127.0.0.0 Foo
время: 04:03:27
ip: 127.0.0.0
company: Foo

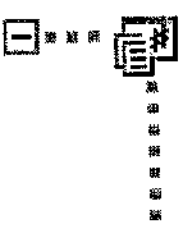
Но ведь это не совсем то, что мы хотели увидеть. Объект theMatch содержит два экземпляра подстроки, соответствующей группе company. Это Double и Foo. Но программа вывела на экран для группы company только одно слово Foo. На самом деле программа добавила в группу company два элемента. Но слово Foo было добавлено в вершину коллекции, поэтому именно оно и вывелось на экран. Если мы хотим увидеть все содержимое группы company, то нам придется переписать код так, чтобы обрабатывались все элементы коллекции.

```
using System;
using System.Text.RegularExpressions;

namespace Programming_CSharp
{
  class Test
  {
    public static void Main( )
    {
      string string1 =
        "04:03:27 Double 127.0.0.0 Foo ";

      Regex theReg = new Regex(
        @"(?<время>(\d|\:)+)\s" +
        @"(?<company>\S+)\s" +
        @"(?<ip>(\d|\.)+)\s" +
        @"(?<company>\S+)\s");

      // получаем коллекцию поиска
      MatchCollection theMatches =
        theReg.Matches(string1);

      // перебор всей коллекции
      foreach (Match theMatch in theMatches)
      {
        if (theMatch.Length != 0)
        {
          // выводим найденную подстроку
          Console.WriteLine("\ntheMatch: {0}",
            theMatch.ToString( ));

          // выводим группу "time"
```

```
            Console.WriteLine("время: {0}",
              theMatch.Groups["время"]);

            // выводим группу "ip"
            Console.WriteLine("ip: {0}",
              theMatch.Groups["ip"]);

            // выводим группу "company"
            Console.WriteLine("company: {0}",
              theMatch.Groups["company"]);

            int idx=0;
            foreach (Capture company in
              theMatch.Groups["company"].Captures)
            {
              idx++;
              Console.WriteLine("company {0}: {1}",
                idx, company.ToString( ));
            }
          }
        }
      }
    }
```

Теперь программа выдаст ожидаемый результат:

theMatch: 04:03:27 Double 127.0.0.0 Foo
время: 04:03:27
ip: 127.0.0.0
company: Foo
company 1: Double
company 2: Foo



Раздел III. Программирование для Windows

19. КНОПКИ И БЛОК ГРУППИРОВКИ

В этом разделе я постарался рассмотреть наиболее часто используемые компоненты при создании Windows Forms приложений. К основным компонентам относятся самые популярные элементы управления Windows, меню, панель управления, с которыми вы постоянно сталкиваетесь при работе в Windows. Примеры программ с использованием описываемых компонентов, приведенные в этой глаЕе, помогут вам быстрее усвоить рассматриваемый материал.

КНОПКИ — Button

Кнопкой называется элемент управления, все взаимодействие пользователя с которым ограничивается одним действием — нажатием. Все, что вам необходимо сделать при работе с кнопкой,— это поместить ее в нужном месте формы и назначить ей соответствующий обработчик. Обработчик назначается для события Click.

ЧЕКБОКСЫ — CheckBox

Первое, что необходимо сказать о чекбоксах, это то, что они являются кнопками отложенного действия, т. е. их нажатие не должно запускать какое-либо немедленное действие. С их помощью пользователи вводят параметры, которые скажутся после, когда действие будет запущено иными элементами управления. Элемент CheckBox может иметь 3 состояния — помеченное, непомеченное и смешанное. Чаще всего этот элемент применяется для определения значений, которые могут иметь только два состояния.

РАДИОКНОПКИ — RadioButton

Радиокнопки по своим свойствам немного похожи на чекбоксы. Их главное различие заключается в том, что группа чекбоксов позволяет выбрать любую комбинацию параметров, радиокнопки же дают возможность выбрать только один параметр. Из этого различия проистекают и все остальные. Например, в группе не может быть меньше двух радиокнопок. Кроме того, у радиокнопок не может быть смешанного состояния (нельзя совместить взаимоисключающие параметры).

БЛОК ГРУППИРОВКИ — GroupBox

GroupBox Блок группировки помогает визуально объединить несколько элементов управления в одну группу. Это бывает особенно полезно, когда надо придать вашему приложению более понятный пользовательский интерфейс. Например, объединить группу радиокнопок (см. рис. 19.1).

Рис. 19.1. Пример использования блока группировки для объединения радиокнопок

Давайте напишем программу, использующую все вышеописанные компоненты. Программа будет выполнять следующие функции: радиокнопки задают текст сообщения, которое будет выводиться по нажатию на обычную кнопку. Чекбокс должен определять — выводить сообщение или нет.

Создайте новый Windows Forms проект под названием TestButtons. Сохраните его в созданную для наших проектов папку. Измените некоторые свойства созданной формы:

Name ~ «TestButtonsForm»

Text — «Тест для кнопок».

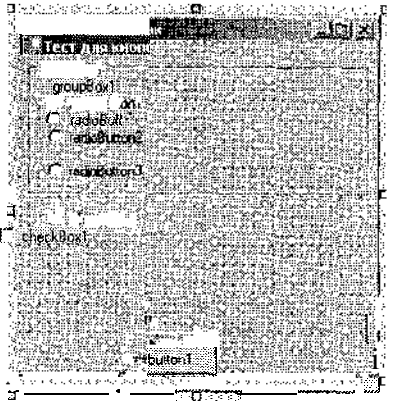


Рис. 19.2. Проектирование формы для приложения TestButtons

Теперь добавьте на вашу форму один элемент управления GroupBox, три элемента RadioButton, один элемент CheckBox и один элемент Button. Я разместил их таким образом, как показано на рис. 19.2. Вы можете последовать моему примеру, или разместить элементы произвольным образом. Однако Заметьте, ЧТО ВСе три раДИОКНОПКИ ДОЛЖНЫ быть помещены в один GroupBox. Иначе они не будут связаны между собой.

Теперь измените некоторые свойства добавленных элементов:

button1:

Text — показать сообщение

groupBox1:

Text — выберите текст сообщения

radioButton1:

Text — первое сообщение

radioButton2:

Text — второе сообщение

radioButton3:

Text — третье сообщение

checkBox1:

Text — показывать сообщение

Checked — True

Возможно, для того чтобы придать вашей форме более достойный вид, вам придется изменить размеры некоторых элементов управления.

Теперь давайте обратимся к коду нашей программы, который создала среда Visual Studio .NET:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace TestButtons
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class TestButtonsForm: System.Windows.Forms.Form
  {
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.GroupBox groupBox1;
    private System.Windows.Forms.RadioButton radioButton1;
    private System.Windows.Forms.RadioButton radioButton2;
    private System.Windows.Forms.RadioButton radioButton3;
    private System.Windows.Forms.CheckBox checkBox1;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public TestButtonsForm()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      if (disposing)
```

```
    {
      if (components != null)
      {
        components.Dispose();
      }
    f
    base.Dispose( disposing );
  }

  #region Windows Form Designer generated code
  /// <summary>
  /// Required method for Designer support - do not modify
  /// the contents of this method with the code editor.
  /// </summary>
  private void InitializeComponent()
  (
    this.button1 = new System.Windows.Forms.Button();
    this.groupBox1 = new System.Windows.Forms.GroupBox();
    this.radioButton3 = new System.Windows.Forms.RadioButton();
    this.radioButton2 = new System.Windows.Forms.RadioButton();
    this.radioButton1 = new System.Windows.Forms.RadioButton();
    this.checkBox1 = new System.Windows.Forms.CheckBox();
    this.groupBox1.SuspendLayout();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(80, 240);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(128, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "Показать сообщение";
    //
    // groupBox1
    //
    this.groupBox1.Controls.AddRange(new System.Windows.Forms.Control[] {
this.radioButton3,
this.radioButton2,
this.radioButton1});
    this.groupBox1.Location = new System.Drawing.Point(8, 16);
    this.groupBox1.Name = "groupBox1";
    this.groupBox1.Size = new System.Drawing.Size(280, 112);
    this.groupBox1.TabIndex = 1;
    this.groupBox1.TabStop = false;
    this.groupBox1.Text = "Выберите текст сообщения";
```

```
//
// radioButton3
//
this.radioButton3.Location = new System.Drawing.Point (16, 80);
this.radioButton3.Name = "radioButton3";
this.radioButton3.Size = new System.Drawing.Size(240, 24);
this.radioButton3.TabIndex = 2;
this.radioButton3.Text = "третье сообщение";
//
// radioButton2
//
this.radioButton2.Location = new System.Drawing.Point(16, 52);
this.radioButton2.Name = "radioButton2";
this.radioButton2.Size = new System.Drawing.Size(240, 24);
this.radioButton2.TabIndex = 1;
this.radioButton2.Text = "второе сообщение";
//
// radioButton1
//
this.radioButton1.Location = new System.Drawing.Point(16, 24);
this.radioButton1.Name = "radioButton1";
this.radioButton1.Size = new System.Drawing.Size(240, 24);
this.radioButton1.TabIndex = 0;
this.radioButton1.Text = "первое сообщение";
//
// checkBox1
//
this.checkBox1.Checked = true;
this.checkBox1.CheckState = System.Windows.Forms.CheckState.Checked;
this.checkBox1.Location = new System.Drawing.Point(24, 136);
this.checkBox1.Name = "checkBox1";
this.checkBox1.Size = new System.Drawing.Size(256, 24);
this.checkBox1.TabIndex = 2;
this.checkBox1.Text = "Показывать сообщение";
//
// TestButtonsForm
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(292, 273);
this.Controls.AddRange(new System.Windows.Forms.Control[] {
                                    this.checkBox1,
                                    this.groupBox1,
                                    this.button1});
this.Name = "TestButtonsForm";
this.Text = "Тест для кнопок";
```

```
        this.groupBox1.ResumeLayout(false);
        this.ResumeLayout(false);

    }
    ttendregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.Run(new Form1());
    }
}
```

Ваш код может отличаться от моего координатами расположения элементов управления и их размером. То есть свойствам Location и Size объектов в функции InitializeComponent могут присваиваться другие значения. Это никак не влияет на работоспособность программы.

Для начала необходимо изменить имя формы, с которой будет запускаться приложение. Как вы помните, мы переименовали Form1 в TestButtonsForm. Найдите в коде строку

```
Application.Run(new Form1());
```

и замените ее на:

```
Application.Run(new TestButtonsForm());
```

Теперь программа работоспособна. Если вы строго следовали всем инструкциям, то программа должна построиться без ошибок. Откомпилируйте ее и запустите. Пока наша программа не способна выполнять какие-либо действия. Давайте наделим ее функциональностью.

Добавьте функцию-обработчик для кнопки button1. Используйте имя по умолчанию для этой функции — button1_Click, создаваемое средой Visual Studio .NET. При этом в функцию InitializeComponent добавится строка:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

и появится тело самой функции:

```
private void button1_Click(object sender, System.EventArgs e)
{

}
```

Добавьте в тело функции button1_Click следующий код:

```
//объявляем строковую переменную
//для хранения выбранного сообщения
string strMessage="";
//определяем какая именно радиокнопка отмечена
//и выбираем в соответствии с этим
//текст выводимого сообщения
```

```
    //проверяем первую радиокнопку
    if(radioButton1.Checked == true)
    {
      //если отмечена именно эта кнопка
      //то копируем текст кнопки в переменную
      strMessage = radioButton1.Text;
    }
    //проверяем вторую радиокнопку
    else if (radioButton2.Checked == true)
    {
      //если отмечена именно эта кнопка
      //то копируем текст кнопки в переменную
      strMessage = radioButton2.Text;
    }
    //проверяем третью радиокнопку
    else if(radioButton3.Checked == true)
    {
      //если отмечена именно эта кнопка
      //то копируем текст кнопки в переменную
      strMessage = radioButton2.Text;
    }

    //проверяем, установлен ли чекбокс
    //разрешающий вывод сообщения
    //если да, то выводим выбранное сообщение на экран
    if (checkBox1.Checked == true)
    {
      MessageBox.Show("Вы выбрали " + strMessage);
    }
  }
```

Подробное описание работы функции приведено в комментариях, поэтому я не буду повторяться. Откомпилируйте и запустите программу. Выберите первую радиокнопку «первое сообщение». Нажмите кнопку *Показать сообщение*. На экране появится надпись: *Вы выбрали первое сообщение*. Выбрав иную радиокнопку, вы получите другой текст сообщения. Теперь уберите флажок *Показывать сообщение*. Нажмите кнопку *Показать сообщение*. На экране ничего не должно появиться. Если программа работает в строгом соответствии с данным описанием, значит, вы все сделали верно.

20. ПОЛЯ ВВОДА И СПИСКИ

ПОЛЕ ВВОДА — TextBox

 Этот элемент управления является основным, предназначенным для ввода пользователем текстовых данных. Использовать TextBox можно в однострочном или многострочном режиме. Однако данный элемент управления имеет ограничение — до 64 кВт текста. Если вам необходимо обрабатывать большие объемы информации, лучше использовать элемент RichTextBox.

РАСШИРЕННОЕ ПОЛЕ ВВОДА — RichTextBox

 Данный элемент управления дает возможность пользователю вводить и обрабатывать большие объемы информации (более 64 кБт). Кроме того, RichTextBox позволяет редактировать цвет текста, шрифт, добавлять изображения. RichTextBox включает все возможности текстового редактора Microsoft Word.

СПИСОК — ListBox

 ListBox — простейший вариант пролистываемого списка. Он позволяет выбирать один или несколько хранящихся в списке элементов. Кроме того, ListBox имеет возможность отображать данные в нескольких колонках. Это позволяет представлять данные в большем объеме и не утомлять пользователя скролированием.

ПОМЕЧАЕМЫЙ СПИСОК — CheckedListBox

 Помечаемый список является разновидностью простого списка. Его дополнительное достоинство — в наличии чекбоксов рядом с каждым элементом списка. Пользователь имеет возможность отметить один или несколько элементов списка, выставив напротив его флажок.

ВЫПАДАЮЩИЙ СПИСОК — ComboBox

 Этот вариант списка удобен тем, что не занимает много пространства на форме. Постоянно на форме представлено только одно зна-

чение этого списка. При необходимости пользователь может раскрыть список и выбрать другое интересующее его значение. Кроме того, режим DropDown дает пользователю возможность вводить собственное значение при отсутствии необходимого значения в списке.

Давайте рассмотрим пример использования списков. Напишем программу, предназначенную для учета данных об участниках соревнований. Программа будет содержать два списка — ComboBox, для ввода информации об участниках, и CheckedListBox, для хранения и обработки данных. С помощью списка ComboBox пользователь будет выбирать фамилии лиц, которых необходимо добавить в список участников. Две кнопки на форме будут добавлять или удалять участников из списка.

Создайте новый Windows Forms проект под названием TestLists. Сохраните его в созданную для наших проектов папку. Переименуйте файл Form1.cs в TestListsForm.cs. Теперь добавьте на вашу форму следующие элементы управления:

- GroupBox, и поместите в него CheckedListBox
- ComboBox
- два элемента Button.

Я разместил их таким образом, как представлено на рис. 20.1. Вы можете последовать моему примеру, или разместить элементы произвольно.

В данном примере можно было обойтись и без элемента GroupBox, так как он предназначен только для оформления интерфейса программы. Однако возьмите себе за хорошую привычку всегда помещать списки внутрь GroupBox элементов. Это сделает ваши программы более привлекательными.

Измените некоторые свойства созданной формы:

Text — «работа со списками»

Теперь изменим свойства элементов управления:

groupBox1:

Text — «список участников»

CheckedListBox:

Name — memberList

comboBox1:

Name — peopleList

Text — «»

button1:

Name — buttonAdd

Text — «Добавить»

button2:

Name — buttonDelete

Text — «Удалить»

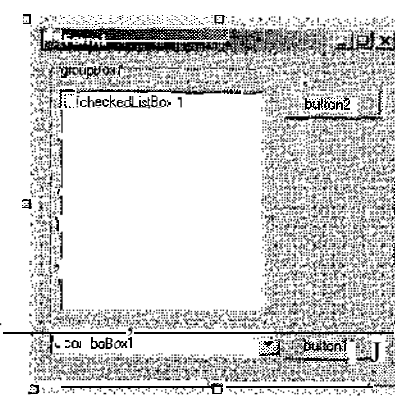


Рис. 20.1. Проектирование формы приложения TestLists

Элементы управления ComboBox и CheckedListBox могут быть проинициализированы с помощью дизайнера среды Visual Studio.Net. Для хранения элементов списков данные компоненты имеют свойство

Items. Свойство Items само по себе является массивом строк. Давайте проинициализируем элемент управления ComboBox, который имеет имя peopleList, списком фамилий предполагаемых участников соревнований. Для этого в окне свойств peopleList выберите свойство Items. Откройте окно *String Collection Editor,* нажав на кнопку с тремя точками в поле Items. Добавьте в предложенный список три фамилии: Иванов, Петров, Сидоров (рис. 20.2).

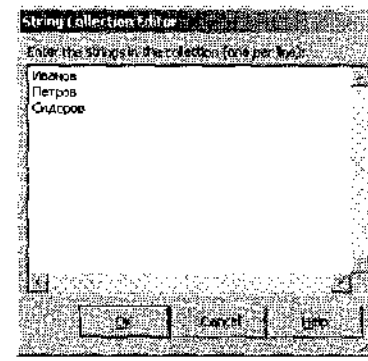


Рис. 20.2. Редактор списка строк

И напоследок, добавьте обработчики для кнопок *Добавить* и *Удалить,* два раза щелкнув левой кнопкой мыши по каждой из кнопок.

Подготовительный этап к написанию программы завершен. Сохраните сделанные вами изменения. Теперь давайте обратимся к коду программы.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace TestLists
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1: System.Windows.Forms.Form
  {
    private System.Windows.Forms.GroupBox groupBox1;
    private System.Windows.Forms.CheckedListBox memberList;
    private System.Windows.Forms.ComboBox peopleList;
    private System.Windows.Forms.Button buttonAdd;
    private System.Windows.Forms.Button buttonDelete;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public TestListsForm()
    {
      //
      // Required for Windows Form Designer support
      //
```

```
        InitializeComponent();

        //
        // TODO: Add any constructor code after InitializeComponent call
        //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
        if( disposing )
        {
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose(disposing);
    }

    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.groupBox1 = new System.Windows.Forms.GroupBox();
        this.memberList = new System.Windows.Forms.CheckedListBox();
        this.peopleList = new System.Windows.Forms.ComboBox();
        this.buttonAdd = new System.Windows.Forms.Button();
        this.buttonDelete = new System.Windows.Forms.Button();
        this.groupBox1.SuspendLayout();
        this.SuspendLayout();
        //
        // groupBox1
        //
        this.groupBox1.Controls.AddRange(new System.Windows.Forms.Control[] {


this.memberList});
        this.groupBox1.Location = new System.Drawing.Point(8, 8);
        this.groupBox1.Name = "groupBox1";
```

```
this.groupBox1.Size = new System.Drawing.Size(184, 216);
this.groupBox1.TabIndex = 0;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "Список участников";
//
// memberList
//
this.memberList.Location = new System.Drawing.Point(8, 24);
this.memberList.Name = "memberList";
this.memberList.Size = new System.Drawing.Size(168, 184);
this.memberList.TabIndex = 0;
//
// peopleList
//
this.peopleList.Items.AddRange(new object[] {
      "Иванов",
      "Петров",
      "Сидоров"});
this.peopleList.Location = new System.Drawing.Point(8, 232);
this.peopleList.Name = "peopleList";
this.peopleList.Size = new System.Drawing.Size(184, 21);
this.peopleList.TabIndex = 1;
//
// buttonAdd
//
this.buttonAdd.Location = new System.Drawing.Point(200, 232);
this.buttonAdd.Name = "buttonAdd";
this.buttonAdd.Size = new System.Drawing.Size(80, 23);
this.buttonAdd.TabIndex = 2;
this.buttonAdd.Text = "Добавить";
this.buttonAdd.Click += new System.EventHandler(this.buttonAdd_Click);
//
// buttonDelete
//
this.buttonDelete.Location = new System.Drawing.Point(200, 32);
this.buttonDelete.Name = "buttonDelete";
this.buttonDelete.Size = new System.Drawing.Size(80, 23);
this.buttonDelete.TabIndex = 3;
this.buttonDelete.Text = "Удалить";
this.buttonDelete.Click += new
System.EventHandler(this.buttonDelete_Click);
//
// TestListsForm
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(292, 273);
```

```
        this.Controls.AddRange(new System.Windows.Forms.Control[] {
                                                this.buttonDelete,
                                                this.buttonAdd,
                                                this.peopleList,
                                                this.groupBox1}) ;
        this.Name = "TestListsForm";
        this.Text = "Работа со списками";
        this.groupBox1.ResumeLayout(false);
        this.ResumeLayout(false);

    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }

    private void buttonDelete_Click(object sender, System.EventArgs e)
    {

    }

    private void buttonAdd_Click(object sender, System.EventArgs e)
    {

    }
  }
}
```

Если вы строго следовали моим инструкциям, то программа должна откомпилироваться и запуститься. Проверьте это.

Сейчас необходимо добавить обработчики для кнопки *«Добавить»* и *«Удалить»*. Как известно из исходных данных, кнопка *«Добавить»* должна заносить строку, выбранную в комбобоксе, в список участников. Для этого измените функцию buttonAdd_Click так, как показано ниже:

```
    private void buttonAdd_Click(object sender, System.EventArgs e)
    {
      //работаем со списком для ввода фамилий
      //проверяем выбран ли элемент в списке
      if(peopleList.Text.Length != 0)
      {
        //если элемент выбран, то переносим его в список участников
```

```
        memberList.Items.Add(peopleList.Text);
      }
      else
      {
        //если элемент не выбран
        //то выдаем информационное сообщение
        MessageBox.Show(
          "Выберите элемент Б списке для ввода или введите новый.");
      }
    }
```

Описание работы функции приведено вместе с ее кодом. Функция memberList.Items.Add добавляет новый элемент в список memberList. При этом параметром функции является значение свойства peopleList.Text, которое выбирает пользователь. Теперь осталось реализовать удаление элементов из списка. Для этого введите код для функции buttonDelete_Click.

```
private void buttonDelete_Click(object sender, System.EventArgs e)
{
  //пока список помеченных элементов не пуст
  while(memberList.CheckedIndices.Count > 0)
  {
    //удаляем из общего списка участников по одному элементу
    //при этом список помеченных элементов автоматически обновляется
    //таким образом, каждый раз нулевой элемент из CheckedIndices
    //будет содержать индекс первого помеченного в списке объекта
    memberList.Items.RemoveAt(memberList.CheckedIndices[0]);

    //при удалении из списка последнего помеченного элемента
    //CheckedIndices.Count станет равным нулю
    //и цикл автоматически завершится
  }
}
```

Функция CheckedListBox.Items.RemoveAt удаляет из списка элемент по его индексу. При этом элементы списка, идущие за удаленным, уменьшают свой индекс на единицу. Это обязательно нужно учитывать при дальнейшем обходе списка.

Класс CheckedListBox содержит свойство CheckedIndices, которое представляет собой массив индексов всех помеченных элементов списка. Этот массив тоже изменяется, если из списка был удален помеченный элемент. А поскольку мы удаляем из списка только помеченные элементы, то CheckedIndices будет изменяться всегда: место удаленного элемента займет следующий за ним. Цикл продолжит работать до тех пор, пока в списке CheckedIndices будет оставаться хоть один элемент.

Давайте рассмотрим работу нашей программы на примере. Откомпилируйте и запустите приложение. Заполните список участников так, как показано на рис. 20.3. Для того чтобы добавить элемент в список, выберите его в комбобоксе и нажмите кнопку Add.

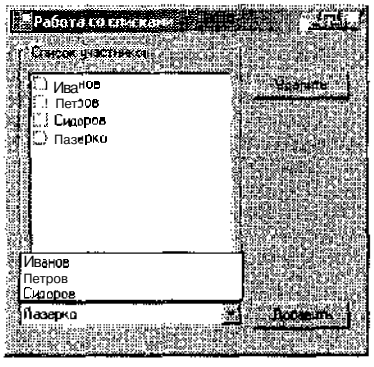


Рис. 20.3. ОКНО приложения TestLists

По умолчанию, элемент управления ComboBox имеет стиль DropDown (свойство ComboBox.DropDownStyle.DropDown). этот стиль дает возможность пользователю не только выбирать элементы из списка, но и вводить данные с клавиатуры. Поэтому для добавления в список фамилии «Лазерко», наберите ее на клавиатуре и нажмите кнопку Add.

Теперь давайте разберемся с логикой работы программы при удалении элементов из списка. Для этого выделите в списке фамилии «Петров» и «Сидоров», которые в нашем списке имеют индексы 1 и 2 соответственно (начиная с нулевого). Поэтому массив CheckedIndices будет содержать два элемента — 1 и 2. При нажатии кнопки *Delete* программа по циклу начинает удалять элементы из списка. Нулевым элементом в массиве CheckedIndices стоит число 1 (индекс фамилии «Петров»), поэтому фамилия «Петров» первой удаляется из списка. При этом фамилии «Сидоров» и «Лазерко» изменяют свои индексы с 2 и 3 на 1 и 2 соответственно. Массив CheckedIndices тоже модифицируется. Во-первых, из него удалится нулевой элемент, индекс фамилии «Петров». Во-вторых, место нулевого элемента в массиве CheckedIndices теперь займет индекс фамилии «Сидоров». А поскольку «Сидоров» теперь имеет индекс 1 в общем списке, то CheckedIndices[0] будет содержать число 1. На второй итерации цикла удаления из списка исчезнет фамилия «Сидоров», а «Лазерко» переместится на позицию 1. В итоге, коллекция CheckedIndices окажется пустой, и цикл завершится.

21. МЕТКИ, ИНДИКАТОРЫ ПРОГРЕССА И БЕГУНКИ

МЕТКА — Label

Элемент управления Label предназначен для создания подписей к другим элементам управления или для вывода информационных сообщений прямо на поверхности формы. Например, вы можете сочетать метки с полями ввода (см. рис. 21.1).

Здесь я расширил нашу форму TestListsForm за счет добавления элемента управления Label с надписью «Выберите фамилию участника или введите новую». Согласитесь, это повышает уровень восприятия программы пользователем.

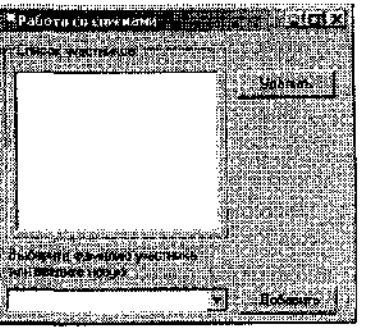


Рис. 21.1. Добавление элемента управления Label к форме приложения TestLists

МЕТКА — LinkLabel

LinkLabel представляет собой гиперссылку, которыми наполнен Интернет. Разработчики Visual Studio .NET представили этот элемент управления как разновидность метки (элемента управления Label). На мой взгляд, LinkLabel более похож на кнопку, чем на метку.

БЕГУНОК — TrackBar

Типичным примером применения элемента TrackBar является регулятор уровня громкости в панели Windows. TrackBar может использоваться в различных режимах: в горизонтальном или вертикальном положении, с включенными черточками или без. Мы рассмотрим далее использование элемента управления TrackBar на примере.

ИНДИКАТОР ПРОГРЕССА — ProgressBar

Чаще всего ProgressBar используют для отображения степени завершенности той или иной задачи. Вы сталкивались с индикатором прогресса, когда устанавливали на свой компьютер Visual Studio .NET.

РЕГУЛЯТОР ЧИСЛЕННЫХ ЗНАЧЕНИЙ — NumericUpDown

Позволяет без помощи клавиатуры вводить численные значения в поле для ввода. Вообще, данный элемент управления имеет три возможности для ввода данных: щелчок мышкой на указатели вверх-вниз, использование кнопок вверх-вниз на клавиатуре или ввод данных в поле ввода.

Для быстрейшего усвоения информации о работе с вышеуказанными компонентами, давайте рассмотрим пример. Напишем приложение, в котором бегунок и элемент управления NumericUpDown управляют индикатором прогресса. Дополнительное условие: бегунок и NumericUpDown должны работать синхронно. То есть, при изменении значения одного элемента, значение другого должно изменятся автоматически на ту же величину.

Создайте новый Windows Forms проект под названием TestIndicator. Сохраните его в созданную для наших проектов папку. Переименуйте файл Form1.cs в TestIndicatorForm.cs. Теперь добавьте на вашу форму следующие элементы управления:

- TrackBar
- ProgressBar
- NumericUpDown.

Мне понравилось размещение, изображенное на рис. 21.2. Вы можете расположить элементы по-своему.

Измените свойства элементов управления.

Свойства элемента TrackBarl:

Maximum — 100

TickStyle — Both.

При этом TrackBar изменит свой вид. Бегунок примет прямоугольную форму, а полоски появятся и сверху, и снизу от него. Это результат изменения свойства TickStyle. Данное свойство определяет месторасположение черточек элемента управления. В этом случае мы выбрали значение Both (с обеих сторон). Кроме того, возможны расположения только сверху, только снизу или вообще без черточек. Свойства Minimum и Maximum задают минимальное и максимальное значение, Д° которых может изменяться TrackBar. В данном случае мы задали максимальное значение 100, а минимальное 0 (оставили по умолчанию). То есть, когда бегунок будет находиться в крайнем левом положении, значение его свойства Value будет равно 0, а при нахождении бегунка в крайнем правом положении свойство Value будет иметь значение 100.

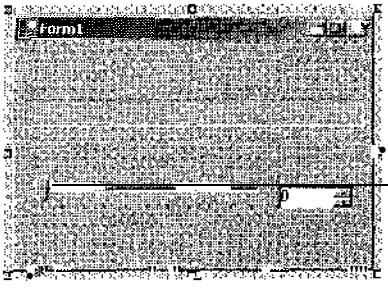


Рис. 21.2. Проектирование формы приложения TestIndicator

Свойства объекта numericUpDown1 оставим по умолчанию. Элемент управления

NumericUpDown также имеет свойства Minimum и Maximum. И по умолчанию, свойство Minimum равно 0, свойство Maximum равно 100. Это соответствует параметрам, установленным для объекта trackBar1. Хочу отметить очень важное свойство компонента NumericUpDown — DecimalPlaces. Оно определяет количество знаков после запятой. В нашем примере это свойство необходимо оставить по умолчанию равным 0, однако при необходимости получить большую точность, чем целое значение, следует устанавливать значение свойства в соответствии с заданной точностью.

Измените значение свойства Text формы на «Управление движением».

Взгляните на код программы. Я не приводил код той части программы, который был построен дизайнером. Он не имеет особого значения, а лишь отражает визуальное содержимое формы в коде программы на языке C#.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace TestIndicator
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1: System.Windows.Forms.Form
  {
    private System.Windows.Forms.TrackBar trackBar1;
    private System.Windows.Forms.ProgressBar progressBar1;
    private System.Windows.Forms.NumericUpDown numericUpDown1;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1()
    {
      // Required for Windows Form Designer support
      InitializeComponent();

      // TODO: Add any constructor code after InitializeComponent call
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
```

```
    {
      if( disposing )
      {
        if (components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
    ...
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
  }
}
```

Как вы уже, наверное, поняли, элементы NumericUpDownl и trackBar1 являются управляющими, а элемент progressBar1 — управляемым. Давайте зададим обработчики событий для управления индикатором прогресса. Итак. Компонент TrackBar имеет событие Scroll, которое предназначено для обработки перемещения указателя бегунка. Создайте функцию обработчик для события Scroll, щелкнув два раза указателем мыши по имени события в окне свойств. В код программы добавится функция с именем trackBar1_Scroll. Измените ее код так, как показано ниже:

```
private void trackBar1_Scroll(object sender, System.EventArgs e)
{
  int Value = trackBar1.Value;
    numericUpDown1.Value = Value;
  progressBar1.Value = Value;
}
```

Теперь при движении курсора бегунка будут изменяться положение индикатора прогресса и значение элемента numericUpDownl. Однако это еще не полная синхронность работы элементов, потому что управление должно вестись из двух элементов: бегунка и числового итератора (NumericUpDown), а у нас сейчас управление возможно лишь от бегунка. Давайте добавим обработчик события ValueChanged для элемента numericUpDownl. Для этого щелкните два раза указателем мыши по имени события

ValueChanged в окне свойств. В код программы добавится функция с именем numericUpDown1_ValueChanged. Измените ее содержимое аналогично функции trackBar1_Scroll.

```
private void numericUpDown1_ValueChanged(object sender, System.EventArgs e)
<
  int Value = (int)numericUpDown1.Value;

  trackBar1.Value = Value;
  progressBar1.Value = Value;
)
```

Все, программа готова к эксплуатации. Откомпилируйте и запустите ее. Попробуйте изменить положение бегунка. При этом индикатор прогресса и числовой итератор изменят свои значения на соответствующие величины. Попробуйте управлять индикатором прогресса при помощи числового итератора. Эффект будет аналогичный работе с бегунком. Попытайтесь найти среднее положение всех элементов управления (рис. 21.3).

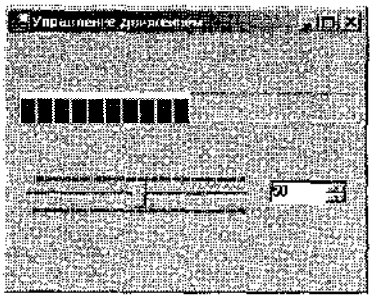


Рис. 21.3. Окно приложения «Управление движением»

22. ListView И TreeView

СПИСОК — ListView

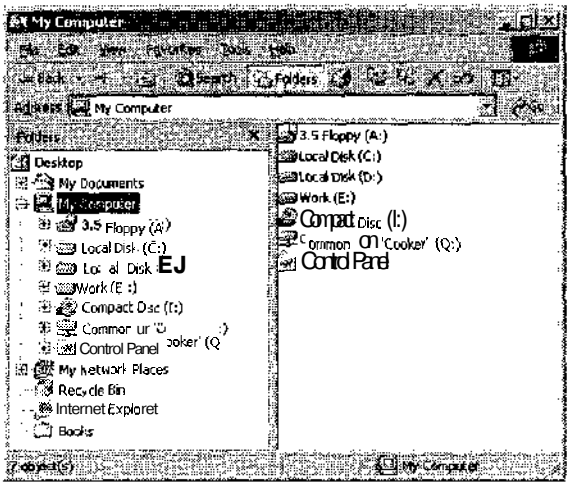


Рис. 22.1. Проводник

ListView Элемент управления ListView предназначен для отображения списков данных. Компонент ListView отличается от ListBox расширенным списком возможностей: установка пиктограмм для каждого элемента списка, отображение данных в нескольких колонках, несколько стилей представления элементов списка. Типичным примером использования элемента ListView является правая сторона программы «Проводник», которая входит в стандартную поставку Microsoft Windows (рис. 22.1).

ДЕРЕВО — TreeView

TreeView Компонент TreeView предназначен для отображения данных в виде дерева. Т. е. элементы представления начинаются с корня дерева и отображаются вглубь. Примером может служить левая сторона программы «Проводник», которая отображает дерево каталогов (см. рис. 22.1).

В левой половине окна располагается содержимое всего диска компьютера. Все элементы списка имеют общий корень «Desktop», то есть все элементы являются подпунктами одного общего корня. В правой стороне отображается содержимое текущей выделенной папки. В моем случае это папка «My Computer». Правая панель отображает элементы в виде списка с пиктограммами.

Давайте создадим приложение, которое будет способно добавлять и удалять элементы в дерево и в список. Главное окно программы будет содержать поле ввода данных, две кнопки: *Добавить в список:* и *Добавить в дерево,* компонент ListView и компонент TreeView. В поле ввода пользователь может набрать любую строку. При нажатии на соответствующую кнопку содержимое поля переносится в ListView или TreeView.

Для этого создайте новое C# Windows-приложение под названием ViewApp. Добавьте на форму следующие элементы:

- ListView;
- TreeView;
- Button — два элемента;
- TextBox.

Я сделал это так, как показано на рис. 22.2. Вы можете построить интерфейс программы по своему усмотрению.

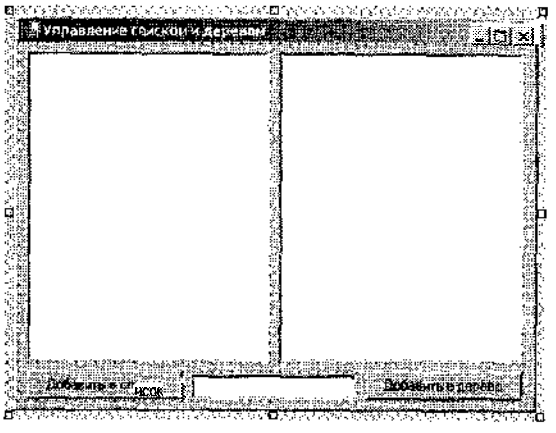


Рис. 22.2. Проектирование формы приложения ViewApp

Измените свойства добавленных компонент:

button1:
    text — «добавить в список»
button2:
    text — «добавить в дерево»
textBox1:
    text — «»

Обратимся к коду программы. (Я не привожу участок кода, сгенерированный дизайнером во время визуального построения формы.)

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace ViewApp
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1: System.Windows.Forms.Form
  {
    private System.Windows.Forms.ListView listView1;
    private System.Windows.Forms.TreeView treeView1;
    private System.Windows.Forms.TextBox textBox1;
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.Button button2;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1()
```

```
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if (components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose (disposing);
    }

    #region Windows Form Designer generated code
    ...
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }

  }
}
```

Работа со списком

Теперь нам предстоит наделить код функциональностью. В первую очередь следует добавить обработчик нажатия кнопки *Добавить в список*. Для этого щелкните два раза указателем мыши по кнопке на фор-

ме. Перед вами откроется окно кода, в который будет добавлена функция button1_Click. Добавьте в функцию button1_Click код, представленный ниже.

```
private void button1_Click(object sender, System.EventArgs e)
{
  // индекс выделенного элемента
  int idx;

  // получаем список всех выделенных объектов
  ListView.SelectedIndexCollection collection = listView1.SelectedIndices;

  // если выделенных объектов нет
  if (collection.Count == 0 )
    idx = 0 ;  // берем нулевой индекс
  // если выделенные объекты есть
  else
    // берем индекс нулевого объекта списка
    idx = collection[0];

  // добавляем новый элемент в список
  listView1.Items.Insert(idx, textBox1.Text);
}
```

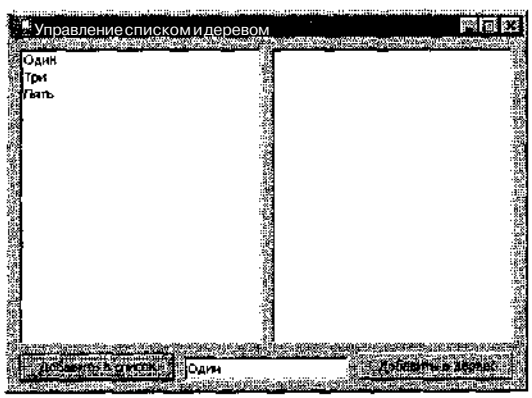


Рис. 22.3. Добавление элементов в список

Откомпилируйте и запустите программу. Выполните описанную ниже последовательность действий: наберите в поле ввода «Пять». Нажмите кнопку *Добавить в список*. Наберите в поле ввода «Три». Нажмите кнопку *Добавить в список*. Наберите в поле ввода «Один». Нажмите кнопку *Добавить в список*. Элементы расположатся в списке так, как показано на рис. 22.3. Каждый новый элемент будет добавлен в начало списка. При этом все предыдущие сместятся вниз.

Теперь выделите в списке пункт «Три». Наберите в поле ввода «Два». Нажмите кнопку *Добавить в список*. Элемент «Два» добавится перед элементом «Три», но после элемента «Один» (там, где он и должен стоять). Это произошло потому, что каждый новый элемент добавляется перед выделенным элементом списка. Выделите элемент «Пять», наберите в поле ввода «Четыре». Нажмите кнопку *Добавить в список*. Элемент «Четыре» добавится между «Три» и «Пять». Давайте рассмотрим код, который заставляет программу работать именно так.

```
ListView.SelectedIndexCollection collection = listView1.SelectedIndices;
```

Класс ListView содержит свойство SelectedIndices со списком индексов всех выделенных элементов списка. Вы можете выделить сразу несколько элементов в списке, удерживая нажатой клавишу *Shift* или *Ctrl*.

```
if(collection.Count == 0 )
  idx = 0;
```

Этот код предназначен для обработки ситуации, в которой ни один элемент списка не выделен. В таком случае элемент будет добавлен в начало списка.

```
else
  idx = collection[0];
```

Если же в списке присутствует хотя бы один индекс, то новый элемент будет добавлен перед самым первым выделенным элементом в списке. Индекс первого выделенного элемента можно получить при помощи выражения collection[0].

```
listView1.Items.Insert(idx, textBox1.Text);
```

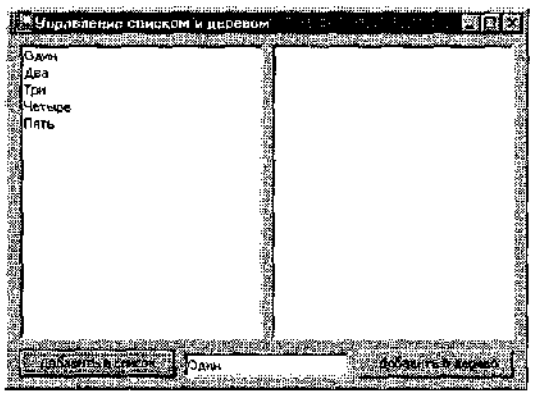


Рис. 22.4. Результаты работы со списком

Эта строка кода добавляет новый элемент в список. Свойство ListView.Items содержит коллекцию всех элементов списка. Функция Insert позволяет добавить новый элемент в список. Новый элемент может быть добавлен в любое место списка. Первый параметр idx указывает позицию нового элемента в списке. Следующий параметр содержит строку для отображения в списке. После выполнения всех вышеописанных операций окно программы будет выглядеть так, как показано на рис. 22.4.

Работа с деревом

Мы рассмотрели, как добавлять элементы в список. Но вторым объектом нашей программы является дерево. Давайте рассмотрим работу с деревом. Для начала необходимо добавить обработчик кнопки *Добавить в дерево*. Для этого щелкните два раза левой кнопкой мыши по кнопке на форме. При этом в программу добавится обработчик нажатия кнопки *button2* — функция button2_Click. Измените код функции button2_Click так, как представлено ниже:

```
private void button2_Click(object sender, System.EventArgs e)
{
      // получаем выделенный узел
      TreeNode node = treeView1.SelectedNode;

      // если выделенного узла нет
```

```
    if (node == null)
    {
      // добавляем новый элемент
      // в корень основного дерева
      treeView1.Nodes.Add(textBox1.Text);
    }
    // если имеется выделенный узел
    else
    {
      // добавляем новый элемент
      // как вложенный в выделенный узел
      node.Nodes.Add(textBox1.Text);
    }
}
```

Откомпилируйте и запустите программу. Выполните описанную ниже последовательность действий. При этом не делайте выделений элементов дерева, пока об этом не будет сказано в описании.

Наберите в поле ввода строку «1-узел», нажмите кнопку *Добавить в дерево*.

Наберите в поле ввода строку «2-узел», нажмите кнопку *Добавить в дерево*.

Наберите в поле ввода строку «3-узел», нажмите кнопку *Добавить в дерево*.

Выделите элемент «1-узел». Наберите в поле ввода строку «1-1-узел», нажмите кнопку *Добавить в дерево*.

Выделите элемент «1-1-узел». Наберите в поле ввода строку «1-1-1-узел», нажмите кнопку *Добавить в дерево*.

Выделите элемент «1-1-узел». Наберите в поле ввода строку «1-1-2-узел», нажмите кнопку *Добавить в дерево*.

У вас должно было получиться такое дерево, как представлено на рис. 22.5.

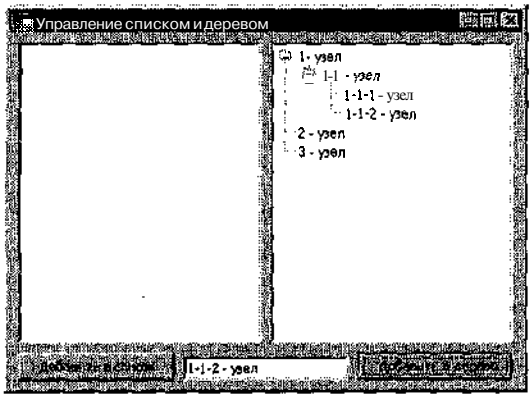


Рис. 22.5. Результаты работы с деревом

Как видите, дерево дает более широкий спектр представления информации. Элементы дерева могут быть встроены в другие элементы. Каждый элемент дерева называется узлом. Если узел содержит другие узлы, то называется корнем дерева. В нашем случае элементы «1-узел» и «1-1-узел» являются корнями для других элементов. Каждый элемент дерева может стать корнем, если встроить в него другие элементы дерева. Корень может содержать сколько угодно элемен-

тов одного уровня. В нашем случае узел «1-1-узел» содержит два элемента одного уровня «1-1-1-узел» и «1-1-2-узел».

Давайте рассмотрим код, который заставляет программу работать именно таким образом:

```
TreeNode node = treeView1.SelectedNode;
```

Класс TreeView имеет свойство SelectedNode. Это свойство возвращает объект TreeNode, который выделен на текущий момент в дереве. Если в дереве не выделен ни один элемент, то свойство SelectedNode возвращает null.

```
if(node == null)
{
treeView1.Nodes.Add(textBox1.Text);
}
```

Если все-таки свойство SelectedNode вернет null, то элемент будет добавлен в список первого уровня. Так были добавлены «1-узел», «2-узел» и «3-узел». Свойство Nodes класса TreeView содержит коллекцию TreeNodeCollection всех узлов первого уровня. Функция Add класса TreeNodeCollection добавляет новый элемент в конец коллекции.

```
else
{
node.Nodes.Add(textBox1.Text);
}
```

Если же один из элементов дерева выделен, то переменная node будет содержать выделенный объект. Класс TreeNode имеет свойство Nodes, которое так же, как и свойство Nodes объекта TreeView, возвращает объект класса TreeNodeCollection. Каждый объект TreeNode содержит коллекцию узлов, для которых он является родительским. Поэтому, когда вы выделили узел «1-узел», новый элемент был добавлен в коллекцию его дочерних узлов. Как и при выделении элемента «1-1-элемент», в коллекцию его дочерних узлов были добавлены элементы «1-1-1-элемент» и «1-1-2-элемент».

Сохраните созданное нами приложение с именем ViewApp, оно нам еще пригодится.

23. СПИОК ИЗОБРАЖЕНИЙ ImageList

ImageList

Компонент ImageList можно отнести к списку невидимых элементов управления. Однако это будет неправильно. Скорее ImageList является вспомогательным компонентом, который хранит изображения, отображаемые другими элементами управления. ImageList используется такими компонентами, как ListView, TreeView, Button, и многими другими. ImageList содержит список изображений, Если какому-либо объекту необходимо отобразить изображение, то необходимо задать для соответствующего свойства лишь индекс элемента в списке изображений.

ImageList может содержать изображения в формате BMP, JPG, GIF, PNG, ICO, WMF, EMF. Для того чтобы построить список изображений, в Visual Studio .NET разработан удобный мастер Image Collection Editor.

ИСПОЛЬЗОВАНИЕ ImageList И ListView

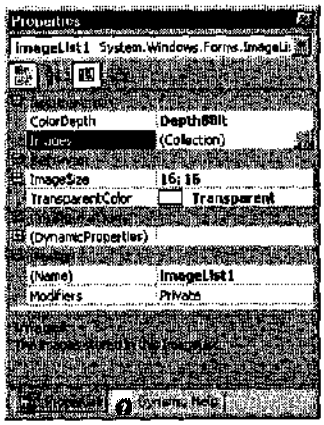


Рис. 23.1. Окно свойств ImageList

Давайте рассмотрим работу с компонентом ImageList на примере. Для этого воспользуемся предыдущим примером.

Откройте приложение ViewApp. Добавьте на форму компонент ImageList. Он отобразится не на форме, а в панели компонент ниже формы. Компонент ImageList имеет свойство Images (рис. 23.1).

Это свойство отвечает за коллекцию изображений, содержащихся в списке. Откройте окно редактирования Image Collection Editor, нажав кнопку с тремя точками в поле Collection окна свойств. Перед вами откроется окно Image Collection Editor (рис. 23.2).

Это окно позволяет добавить новое изображение в коллекцию или удалить уже существующее. Для того чтобы добавить новое изображение, вам необходимо нажать кнопку *Add*. При этом Visual Studio .NET предложит вам выбрать файл, содержащий изображение в одном из доступных форматов. Вы можете выбрать любой файл, по своему усмотрению. Я взял самый близко расположенный файл пиктограммы «App.ico», созданный вместе с каркасом приложения. При этом миниатюр-

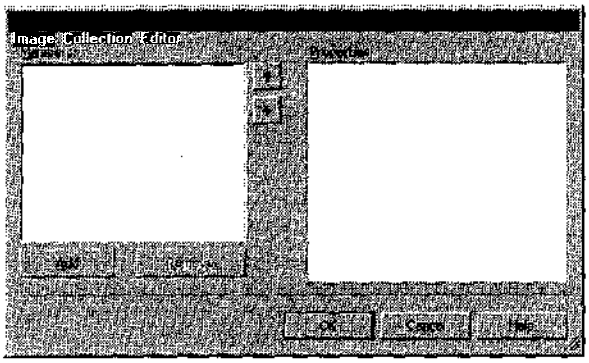


Рис. 23.2. Окно настройки коллекции изображений

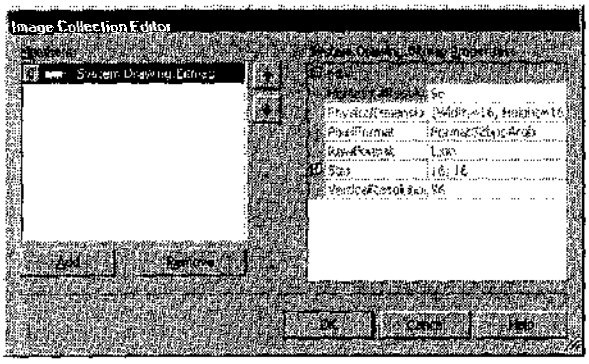


Рис. 23.3. Добавление нового изображения в список

ное изображение пиктограммы появится в списке Member окна (рис. 23.3).

Список Members может содержать сколько угодно элементов. Для нашего примера будет достаточно одного элемента в списке. Закройте Image Collection Editor, нажав кнопку *OK*.

Компонент ListView имеет свойство SmallImageList, которое хранит список изображений, закрепленный за ListView объектом. Установите в окне свойств объекта imageList1 свойство SmallImageList как imageList1. Теперь за списком listView1 будет закреплен список изображений imageListl.

Далее давайте изменим код нашей программы, для того чтобы список мог содержать не только текстовую информацию, но и графическую. Программа потребует минимальных изменений. Класс ListViewItemsCollection содержит несколько вариантов функции Insert. Мы использовали Insert(int index, string text). Другим вариантом функции является Insertfint index, string text, int imageIndex). Третьим параметром этой функции является индекс изображения в списке ImageList, которое было предварительно установлено для ListView.

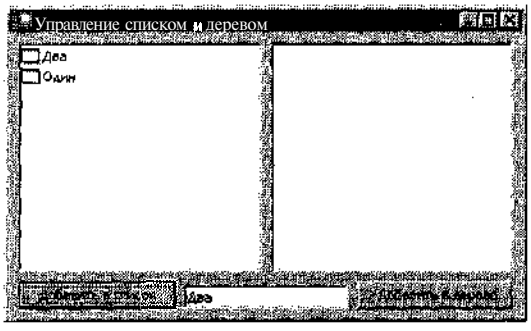


Рис. 23.4. Отображение пиктограмм с элементами списка

Замените вызов функции

```
listView1.Items.Insert(idx,
textBox1.Text);
```

внутри функции button1_Click на

```
listView1.Items.Insert(idx,
textBox1.Text, 0);
```

Добавился лишь третий параметр при вызове функции. Однако попробуйте вновь запустить программу. Добавьте в список элементы «Один» и «Два». На этот раз список бу-

дет выглядеть гораздо симпатичнее. Вместе с текстовым представлением элемента на экране будет отображаться пиктограмма (рис. 23.4).

Для всех элементов нашего списка пиктограмма будет одна и та же. Однако вы можете установить в качестве пиктограммы любой элемент списка изображений. Третьим параметром функции Insert является индекс изображения из списка ImageList. Если ImageList содержит пять элементов, то для установки пиктограммы с индексом 3 вам необходимо вызвать функцию Insert как:

```
listViewl.Items.Insert(idx,textBoxl.Text, 2) ;
```

ИСПОЛЬЗОВАНИЕ ImageList И TreeView

Так же, как пиктограммы в ListView компоненте, пиктограммы могут использоваться другими компонентами. Использование ImageList схоже в применении со всеми компонентами. Вот как выглядят ImageList вместе с TreeView.

В окне свойств объекта treeViewl установите свойство ImageList в imageList1.

Для того чтобы добавить отображение пиктограммы в дерево, измените код в функции button2_Click. Замените вызов функции

```
node.Nodes.Add(textBoxl.Text);
```

на следующий код:

```
TreeNode newNode = new TreeNode ();
newNode.Text = textBox1.Text;
newNode.ImageIndex = 0;
node.Nodes.Add(newNode);
```

Функция Add класса TreeNodeCol lection так же перегружена, как и функция Insert класса ListViewItemsCollection. Вторым вариантом функции Add является Add(System.Windows.Forms.TreeNode node). Для того чтобы добавить текстовый элемент с пиктографическим изображением в дерево, необходимо:

- установить у TreeView свойств ImageList;
- создать элемент TreeNode;
- установить свойство Text;
- установить свойство ImageIndex;
- вызвать функцию TreeNode.Nodes.Add().

Запустите программу с новым кодом. Попробуйте добавить элементы в дерево. Так же, как и в списке, все узлы дерева будут отображаться с пиктограммой (рИС. 23.5).

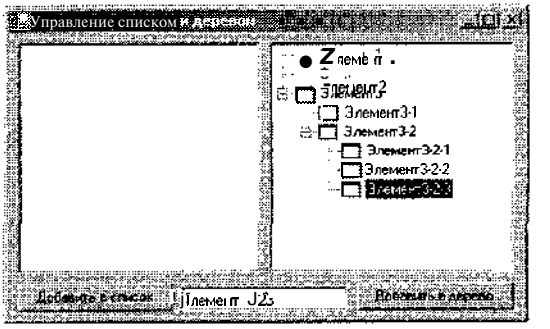


Рис 23.5Использование пиктограмм компонентом TreeView

24. ПОЛОСЫ ПРОКРУТКИ

ОБЩИЕ СВЕДЕНИЯ

Горизонтальные и вертикальные полосы прокрутки широко используются в приложениях Windows. Они обеспечивают интуитивный способ передвижения по спискам информации и позволяют делать поля для ввода данных очень большими.

Полоса прокрутки состоит из трех областей, которые нажимаются или перемещаются для изменения значения полосы прокрутки.

Рассмотрим работу элемента на примере горизонтальной полосы прокрутки. Нажатие левой стрелки уменьшает значение положения бегунка на минимальное. Нажатие правой стрелки увеличивает значение положения бегунка на минимальное. При нажатии курсором мыши в области полосы прокрутки значение положения бегунка изменяется на величину, большую, чем значение при нажатии на кнопки «влево» и «вправо». При использовании реквизитов полос прокрутки мы можем полностью определять, как работает каждый из них. Позиция бегунка — единственная выходная информация из полосы прокрутки.

СВОЙСТВА ПОЛОС ПРОКРУТКИ

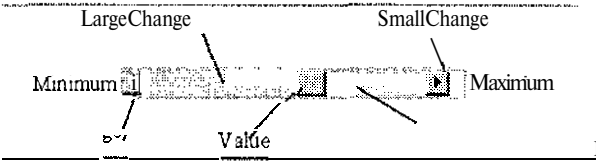


Рис. 24.1. Свойства полос прокрутки

LargeChange — приращение, значение, которое добавляется или вычитается из значения текущего положения бегунка; это величина, на которую изменяется положение бегунка при нажатии курсора в области полосы прокрутки.

Maximum — значение горизонтальной полосы прокрутки в крайнем правом положении и значение вертикальной полосы прокрутки в крайнем нижнем положении. Может принимать значения от -32,768 до 32,767.

Minimum — другое предельное значение — для горизонтальной полосы прокрутки в крайнем левом положении и для вертикальной полосы прокрутки в крайнем верхнем. Может принимать значения от –32,768 до 32,767.

SmallChange — приращение, значение, которое добавляется или вычитается из значения текущего положения бегунка: значение, на которое изменяется положение полосы прокрутки, когда нажата любая из стрелок прокрутки.

Value — текущая позиция бегунка внутри полосы прокрутки. Если вы устанавливаете это в коде, то визуальное положение бегунка перемещается в соответствующую позицию.

СОБЫТИЯ ПОЛОСЫ ПРОКРУТКИ

Change — событие, генерируемое после того, как позиция бегунка изменилась. Используйте это событие, чтобы считать новое значение положения бегунка после любых изменений в полосе прокрутки.

Scroll — событие, вызываемое непрерывно всякий раз, когда бегунок передвигается.

Создайте новый Windows Forms проект с именем ScroUApp. Поместите на форму элемент VScrollBar и четыре элемента Label. Сделайте это так, как показано на рис. 24.2.

Установите для компонент свойства в соответствии с приведенным ниже списком:

Form1:
- Text — измерение температуры

vScrollBar1:
- LargeChange — 10
- Maximum — 60
- Minimum — -20
- SmallChange — 1
- Value — 32

label1:
- Text — Фаренгейт
- Font Size — 10
- Font Bold — True

label2:
- BackColor — White
- Text — 32
- Font Size — 14
- Font Bold — True
- Name — labelFarTemp

label3:
- Text — Цельсий
- Font Size — 10
- FontBold — True

label4:
- BackColor — White
- Text — 0
- Font Size — 14
- Font Bold — True
- Name — labelCTemp

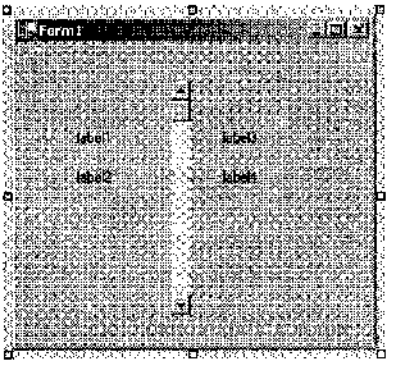


Рис. 24.2. Проектирование формы приложения ScroUApp, этап 1

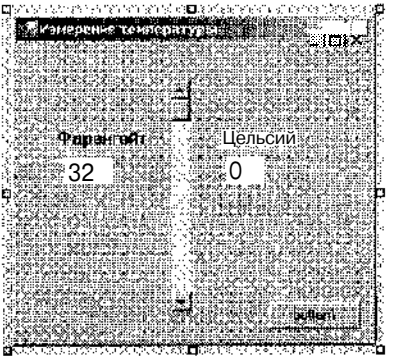


Рис. 24.3. Проектирование формы приложения ScroUApp, этап 2

Обратите внимание, что значения температур проинициализированы 32F и 0C. Когда вы сделаете свою форму, она должна иметь вид, представленный на рис. 24.3.

Как уже отмечалось, компонент ScrollBar имеет событие Scroll. Давайте добавим обработчик этого события в код программы. Для этого в окне свойств на закладке событий для элемента vScrollBar1 щелкните два раза по событию Scroll. В код программы добавится обработчик события Scroll:

```
this.vScrollBar1.Scroll += new
System.Windows.Forms.ScrollEventHandler(this.vScrollBar1_Scroll);
```

и тело функции обработчика. Добавьте код функции vScrollBar1_Scroll, представленый ниже:

```
private void vScrollBar1_Scroll (object: sender,
  System.Windows.Forms.ScrollEventArgs e)
{
  labelFarTemp.Text = vScrollBar1.Value.ToString();
  labelCTemp.Text =
    Convert.ToString((int) ((double)vScrollBar1.Value-32)/5*9));
}
```

Этот код переводит значение Value полосы прокрутки во время изменения положения бегунка и определяет значение температуры по шкале Фаренгейта. Затем вычисляется значение температуры по Цельсию и отображаются оба значения.

Откомпилируйте и запустите программу. Вы увидите, что при изменении положения бегунка полосы прокрутки изменяются значения температур (рис. 24.4).

Попробуйте найти точку, в которой значение температуры по Цельсию равно значению температуры по Фаренгейту.

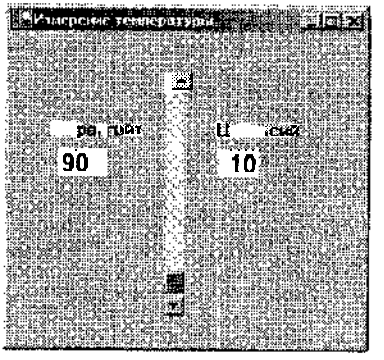


Рис. 24.4. Окно программы ScrollApp

25. МЕНЮ

Обычно программы в среде Windows имеют головное меню, которое образует более или менее сложную структуру выполняемых приложением команд. Создание головного меню приложения является фактически альтернативой выполнения команд элементами управления на форме в соответствии с программным кодом процедур, написанных для этих элементов. Однако, в отличие от элементов управления, которые осуществляют выполнение различных процедур при совершении различных событий над элементом управления, головное меню приложения позволяет создавать иерархию вложенных друг в друга меню команд любой степени сложности.

Особенно эффективно создание головного меню приложения в том случае, когда в приложении необходимо выполнять множество различных команд. При этом команды, выполняющиеся из головного меню, будут собраны в одной строке головного меню, которое можно раскрыть в любой нужный момент для выполнения требуемой команды, не занимая много места в окне приложения на экране дисплея, где должна происходить основная обработка информации.

СОЗДАНИЕ ГОЛОВНОГО МЕНЮ

Для создания головного меню приложения Visual Studio .NET имеет в панели ToolBox компонент MainMenu. Создайте новое Windows Forms C# приложение с именем MenuApp. Добавьте на форму компонент MainMenu. В панели компонентов ниже основной формы приложения появится объект mainMenu1. В верхней части формы появится проект меню с единственным полем «Type Here». Поле является редактируемым, если вы измените надпись в поле, то справа и снизу от него появятся дополнительные поля (рис. 25.1).

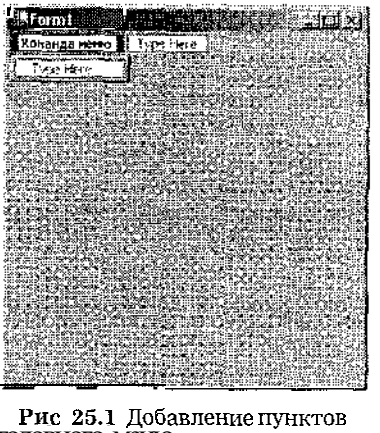


Рис 25.1 Добавление пунктов головного меню

Добавьте в меню пункты так, как показано на рис. 25.2.

Пункт меню «Команда меню» содержит три подпункта:

- Добавить
- Удалить
- Переместить.

Кроме пункта «Команда меню» основное меню содержит пункт «О программе». Такой пункт обычно присутствует во всех коммерчес-

ких приложениях и предоставляет пользователю информацию о версии программы, разработчиках и т. д. Пункт меню «О программе» не имеет подпунктов. Для лучшей читаемости программы измените свойство Name каждого пункта меню.

Команда меню — menuItemCommand;
Добавить — menuItemAdd;
Удалить — menuItemDel;
Переместить — menuItemMove;
О программе — menuItemAbout.

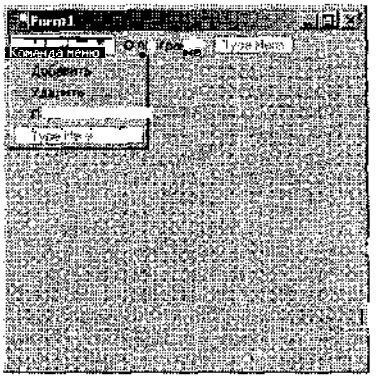


Рис. 25.2. Основное меню приложения MenuApp

Компонент MainMenu представлен в коде программы классом System.Windows.Forms.MainMenu.

```
private System.Windows.Forms.MainMenu mainMenu1;
```

Каждому пункту меню в коде программы соответствует объект класса MenuItem.

```
private System.Windows.Forms.MenuItem menuItemCommand;
private System.Windows.Forms.MenuItem menuItemAdd;
private System.Windows.Forms.MenuItem menuItemDel;
private System.Windows.Forms.MenuItem menuItemMove;
private System.Windows.Forms.MenuItem menuItemAbout;
```

Вот как происходит добавление элементов головного меню:

```
this.mainMenu1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] (
 this.menuItemCommand,
 this.menuItemAbout});
```

Эти строки кода добавляют в объект MainMenu два пункта меню: «Команда меню» и «О программе». Свойство MenuItems объекта MainMenu имеет функцию AddRange, которая добавляет массив элементов MenuItem в меню.

```
this.menuItemCommand.MenuItems.AddRange(
 newSystem.Windows.Forms.MenuItem[] {
 this.menuItemAdd,
 this.menuItemDel,
 this.menuItemMove});
```

Затем, в пункт меню menuItemCommand («Команда меню») добавляется три пункта menuItemAdd, menuItemDel, menuItemMove.

Теперь становится все очень понятно: для того чтобы внести изменения в какой-либо пункт меню, вам необходимо будет изменить соответствующий объект класса MenuItem.

СОЗДАНИЕ ВЛОЖЕННОГО МЕНЮ

Разработчики C# максимально упростили работу с меню. Поэтому создание вложенного меню не вызовет у вас затруднений. Скажу больше: мы уже создавали вложенное меню, когда добавляли в пункт меню «Ко-

манда меню» несколько пунктов. Но там мы помещали подпункты в головное меню приложения. Подпункты меню тоже могут содержать вложенные элементы. Для того чтобы «вложить» элемент в пункт меню, необходимо добавить название в поле «Type Here», находящееся справа от пункта меню (рис. 25.3).

При этом с правой стороны пункта меню появится указательная стрелка, свидетельствующая о наличии вложенных пунктов меню. Добавьте в наше приложение пункт меню «Уведомить» и вложите в него пункты «Сообщение 1», «Сообщение 2», «Сообщение 3» так, как показано на рис. 25.3.

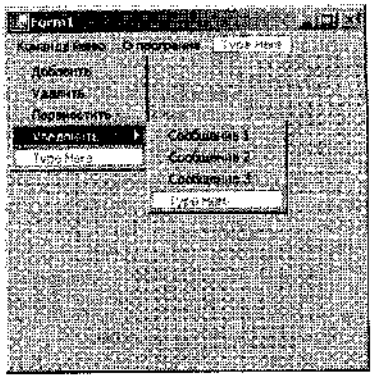


Рис. 25.3. Создание вложенного меню

ОБРАБОТКА СООБЩЕНИЙ МЕНЮ

Основным сообщением пункта меню является Click. Это сообщение приходит, когда пользователь выбирает пункт меню. Давайте добавим обработчики к нашим пунктам меню. Для этого необходимо щелкнуть два раза по полю с именем сообщения в окне свойств пункта меню (рис. 25.4).

Добавьте обработчики для пунктов меню «Добавить», «Удалить» и «Переместить». При этом в код программы добавятся три новых метода. Измените код этих методов так, как показано ниже:

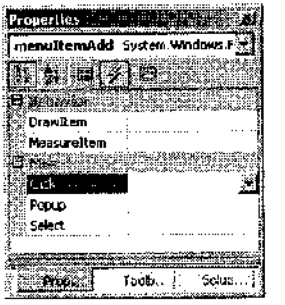


Рис. 25.4. Окно свойств элемента MenuItem. Закладка событий

```
private void menuItemAdd_Click(object sender, System.EventArgs e)
{
  // код для добавления

  MessageBox.Show("Добавление");
}
private void menuItemDel_Click(object sender, System.EventArgs e)
{
  // код для добавления

  MessageBox.Show("Удаление");
}
private void menuItemMove_Click(object sender, System.EventArgs e)
{
  // код для перемещения
  //...
```

```
    MessageBox.Show("Перемещение");
}
```

Теперь при выборе одного из указанных выше пунктов меню на экране появится соответствующее сообщение.

Кроме того, C# позволяет нескольким пунктам меню использовать один и тот же обработчик сообщения. Для того чтобы проэкспериментировать с такой возможностью, выделите сразу три пункта меню: «Сообщение 1», «Сообщение 2», «Сообщение 3», удерживая нажатой клавишу Ctrl. Теперь в окне свойств щелкните два раза по событию Click. Таким образом, вы присвоите всем трем пунктам меню один и тот же обработчик. Посмотрите, как это выглядит в коде программы:

```
this.menuItem2.Click += new System.EventHandler(this.menuItem2_Click);
this.menuItem3.Click += new System.EventHandler(this.menuItem2_Click);
this.menuItem4.Click += new System.EventHandler(this.menuItem2_Click);
```

Для всех трех событий различных пунктов меню устанавливается один и тот же обработчик — menuItem2_Click. Напишите следующий код для функции menuItem2_Click:

```
private void menuItem2_Click(object sender, System.EventArgs e)
{
  MenuItem item = (MenuItem)sender;
  string message = item.Text;
  MessageBox.Show(message);
}
```

Здесь я позволю себе раскрыть один из загадочных, наверное, для вас параметров любого обработчика события. object sender — это объект, который послал сообщение. Поскольку все объекты в C# являются наследниками класса System.Object, то использование класса object в качестве параметра «универсализирует» использование обработчиков событий. В нашем случае мы знали, что обработчик menuItem2_Click получает события только от объектов класса Menultem. Поэтому мы можем смело приводить объект sender к классу Menultem.

```
MenuItem item = (MenuItem)sender;
```

Для того чтобы различить, от какого именно пункта меню пришло событие, мы читаем значение свойства Text пункта меню.

```
string message = item.Text;
```

Это свойство различно у всех пунктов меню. Поэтому мы можем вычислить, какой именно объект прислал сообщение. В нашем случае, мы просто выводим на экран сообщение с текстом пункта меню.

```
MessageBox.Show(message);
```

КОНТЕКСТНОЕ МЕНЮ

Головное и контекстное меню — это, на мой взгляд, абсолютно различные вещи с точки зрения функционального назначения. В головное меню выносят все функции, которые выполняет программа. В любой момент

пользователь может воспользоваться нужным пунктом меню для совершения какого-либо действия. С контекстным меню все по-другому. Оно должно включать лишь те пункты, которые соответствуют позиции вызова контекстного меню. Контекстное меню появляется при нажатии правой кнопки мыши.

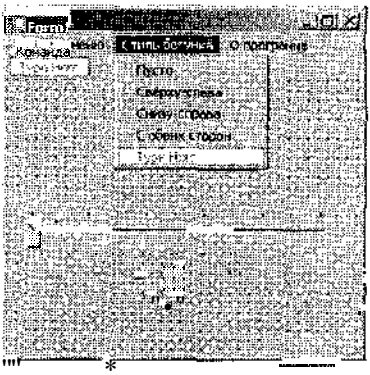


Рис. 25.5. Головное меню управления стилем бегунка

Давайте рассмотрим это на примере. Поместите на форму приложения MenuApp компонент TrackBar. Расположите его по своему усмотрению. В головное меню приложения между пунктами «Команда меню» и «О программе» добавьте пункт «Стиль бегунка». Вложите в этот пункт четыре подпункта: «Пусто», «Сверху-слева», «Снизу-справа», «С обеих сторон» (рис. 25.5).

Переименуйте пункты меню, изменив свойство Name каждого элемента:

- Стиль бегунка — menuItemTrackBar;
- Пусто — menuItemNone;
- Сверху-слева — menuItemTopLeft;
- Снизу-справа — menuItemBottomRight;
- С обеих сторон — menuItemBoth.

Для всех вышеописанных элементов меню присвойте один и тот же обработчик сообщения. Для этого выделите по очереди все четыре пункта мешо, удерживая нажатой кнопку Ctrl. В окне свойств, на закладке Properties, щелкните два раза указателем мыши по событию Click. При этом ко всем пунктам меню добавится обработчик события menuItemNone_Click[1].

```
this.menuItemNone.Click += new System.EventHandler(this.menuItemNone_Click);
this.menuItemTopLeft.Click <= new System.EventHandler(this.menuItemNone Click);
this.menuItemBottomRight.Click += new
System.EventHandler(this.menuItemNone_Click);
this.menuItemBoth.Click += new System.EventHandler(this.menuItemNone Click);
```

В конец кода программы добавится тело самой функции. Теперь давайте создадим два контекстных меню. Одно из них будет частично дублировать пункт основного меню «Команда меню», другое — пункт «Стиль бегунка». Для этого поместите на форму два компонента ContextMenu: contextMenul и contextMenu2. Выберите в панели компонент программы объект contextMenul. При этом в верхней части формы появится редактируемое поле. Выделите его указателем мыши. Ниже выделенного поля появится поле ввода с надписью «Type Here». Добавьте в контекстное меню пункты «Добавить», «Удалить» и «Переместить». Это контекстное

[1] Возможно, имя обработчика события будет называться по-другому. Я не нашел зависимости при экспериментировании с добавлением этого обработчика. В любом случае, имя обработчика должно соответствовать одному из идентификаторов пунктов меню. Не имеет значения, какому именно, просто в вашей версии программы будет другое имя функции-обработчика.

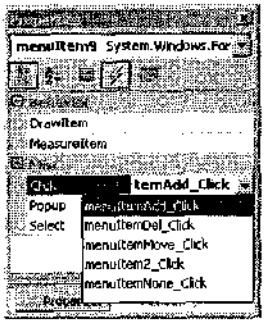


Рис. 25.6. Доступные обработчики событий

меню будет соответствовать пункту основного меню «Команда меню».

Теперь необходимо присвоить обработчики всем добавленным пунктам меню. Для пункта меню «Добавить» выберите из списка обработчиков события Click функцию menuItemAdd_Click (рис. 25.6).

Таким образом, событию Click пункта контекстного меню «Добавить» присвоится обработчик события Click пункта основного меню «Добавить».

```
this.menuItem9.Click += new
System.EventHandler(this.menuItemAdd_Click);
```

Тело функции-обработчика уже существует, поэтому при выборе любого из этих двух пунктов меню будет выполняться одно и то же действие. Установите для пунктов контекстного меню contextMenu1 «Удалить» и «Переместить» обработчики menuItemDel_Click и menuItemMove_Click соответственно. После того как вы это сделаете, необходимо установить contextMenu1 контекстным меню для формы Form1. Для этого у объекта Form1 существует свойство ContextMenu. Visual Studio .NET сама добавит в список ContextMenu все необходимые элементы, которые могут быть присвоены этому свойству. В нашем случае список будет состоять из двух элементов (рис. 25.7).

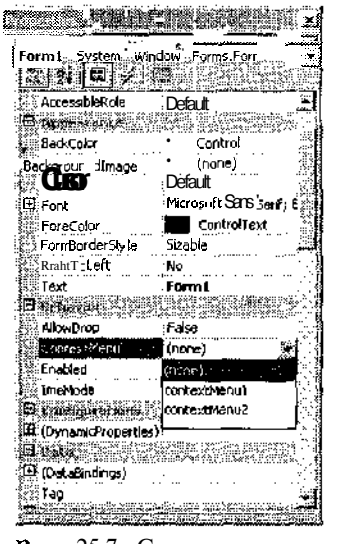


Рис. 25.7. Список доступных контекстных меню

Установите значение свойства ContextMenu в contextMenu1. Откомпилируйте и запустите программу. Щелкните правой кнопкой мыши по любому свободному месту на форме. На экране появится контекстное меню, которое вы создали (рис. 25.8).

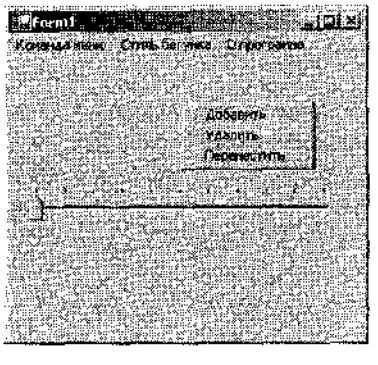


Рис. 25.8. Контекстное меню основной формы программы

Если вы выберите любой из пунктов контекстного меню, выполнится то же действие, которое соответствует аналогичным пунктам основного меню. Выберите пункт «Удалить» контекстного меню. На экране появится сообщение, изображенное на рис. 25.9. Такое же сообщение появится, если вы выберете пункт «Удалить» основного меню.

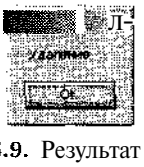


Рис. 25.9. Результат команды «Удалить»

Как я уже отмечал, контекстное меню — это удобная возможность выполнять различные дей-

ствия над объектами, не загромождая основное окно программы. Допустим, нам необходимо в программе изменять свойства элемента TrackBar. Предоставим пользователю возможность делать это как из основного меню программы, так и из контекстного меню. В основном меню программы у нас уже имеется пункт «Стиль бегунка», который содержит подпункты с названиями стилей. Давайте добавим в контекстное меню contextMenu2 те же подпункты. Для этого выделите объект contextMenu2 и добавьте в него поля: «Пусто», «Сверху-слева», «Снизу-справа», «С обеих сторон». Я думаю, это не вызовет у вас трудностей.

Теперь добавьте обработчик события Click для пунктов контекстного меню. Для этого выделите все четыре пункта меню, удерживая клавишу Ctrl, и выберите из предложенного списка для события Click обработчик menuItemNone_Click. Теперь все пункты contextMenu2 и соответствующие пункты основного меню имеют один и тот же обработчик — menuItemNone_Click. Добавьте к этому обработчику код, который представлен ниже:

```
private void menuItemNone_Click (object sender, System.EventArgs e)
{
  MenuItem item = (MenuItem)sender;
  string text = item.Text;
  switch(text)
  {
    case "Пусто":
      trackBar1.TickStyle = TickStyle.None;
      break;
    case "Сверху-слева":
      trackBar1.TickStyle = TickStyle.TopLeft;
      break;
    case "Снизу-справа":
      trackBar1.TickStyle = TickStyle.BottomRight;
      break;
    case "С обеих сторон":
      trackBar1.TickStyle = TickStyle.Both;
      break;
  }
}
```

Для завершения функциональности программы установите свойство ContextMenu компонента trackBarl как contextMenu2. Все, программа закончена. Откомпилируйте и запустите программу. Щелкните правой кнопкой мыши по изображению бегунка на форме.

Вместо контекстного меню, которое появляется при нажатии правой кнопкой мыши на форме, вы увидите контекстное меню, соответствующее управлению стилями бегунка (рис. 25.10).

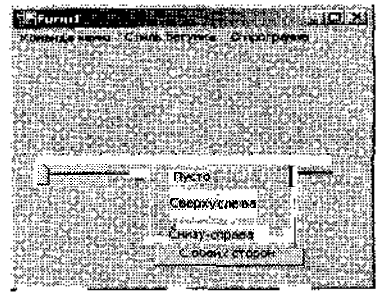


Рис 25 10 Контекстное меню бегунка

Попробуйте изменить стиль бегунка путем выбора различных пунктов контекстного меню. Местоположение черточек бегунка будет изменяться в зависимости от выбранного пункта меню. Те же самые операции можно выполнить из основного меню приложения.

Давайте детальнее рассмотрим код функции-обработчика.

```
MenuItem item = (MenuItem)sender;
```

Присваиваем переменной item объект sender, от которого пришло сообщение. Поскольку заранее известно, что сообщения всегда посылаются от пунктов меню, выполняем приведение типа к MenuItem. Объекты класса MenuItem имеют свойство Text, которое в нашем случае характеризует пункт меню. Инструкция switch в языке C# поддерживает использование строк в качестве элементов для сравнения (глава 7). Поэтому мы получаем значения свойства Text и по нему идентифицируем пункт меню.

```
string text = item.Text;
switch(text)
```

Далее выполняются несколько инструкций case, и по тексту пункта меню выставляется стиль элемента управления TrackBar.

```
case "Пусто":
  trackBar1.TickStyle = TickStyle.None;
  break;
case "Сверху-слева":
  trackBar1.TickStyle = TickStyle.TopLeft;
  break;
case "Снизу-справа":
  trackBar1.TickStyle = TickStyle.BottomRight;
  break;
case "С обеих сторон":
  trackBar1.TickStyle = TickStyle.Both;
  break;
```

ПОМЕТКА ПУНКТОВ МЕНЮ

Рис. 25.11. Пометка пунктов меню

Очень удобная возможность пунктов меню — их пометка. Обычно это флажок слева от надписи пункта меню (рИС. 25.11).

На этом рис. изображены пункты меню программы Paint, входящей в поставку Microsoft Windows. Пункты меню «Tool Box», «Color Box», «Status Bar», «Text Toolbar» предназначены для отображения и скрытия различных панелей программы. Если панель видна, то соответствующий пункт меню помечен флажком, «птичкой» слева от надписи.

Для пометки пункта меню используется свойство Checked класса Menultem. Это свойство типа book Если значение Checked установлено в true, то флажок присутствует, если в false — то отсутствует.

Давайте допишем программу так, чтобы текущий стиль элемента управления TrackBar trackBarl в меню всегда был отмечен флажком. Для этого необходимо изменить код программы. Во-первых, необходимо добавить в описание класса Form1 определение функции menuItemCheck:

```
private void menuItemCheck(string text)
{
  // проходим по всем подпунктам изменяющим стиль бегунка
  // расположенным в основном меню программы
  foreach(MenuItem item in menuItemTrackBar.MenuItems)
  {
    // если текст меню совпадает с переданным параметром
    // то помечаем пункт меню
    if(item.Text == text)
    {
      item.Checked = true;
    }
    // если текст меню не совпадает с переданным параметром
    // то снимаем пометку с пункта меню
    else
    {
      item.Checked = false;
    }
  }

  // проходим по всем подпунктам изменяющим стиль бегунка
  // расположенных в основном меню программы
  foreach(MenuItem item in contextMenu2.MenuItems)
  {
    // если текст меню совпадает с переданным параметром
    // то помечаем пункт меню
    if(item.Text == text)
    {
      item.Checked = true;
    }
      // если текст меню не совпадает с переданным параметром
    // то снимаем пометку с пункта меню
    else
    {
      item.Checked = false;
    }
  }
}
```

Далее необходимо добавить вызов этой функции в функцию menuItemNone_Click. Поместите в конец функции menuItemNone_Click следующую строку кода:

```
menuItemCheck(text);
```

Таким образом, мы вызовем функцию установки флажков для пунктов меню, передав ей в качестве параметра текст выбранного пункта меню.

Еще один маленький штришок. Для того чтобы программа с самого начала правильно функционировала, установите для пунктов меню «Сверху-слева» свойство Checked в True по умолчанию, потому как именно этот стиль установлен по умолчанию для объекта trackBar1. Вы можете сделать это, выбрав нужный пункт меню и установив для него свойство Checked как True в окне свойств.

Запустите программу. Выберите в контекстном меню стиль «С обеих сторон». Откройте еще раз контекстное меню — пункт меню «С обеих сторон» будет помечен. Можете изменить стиль на любой другой, пометка всегда будет соответствовать выбранному стилю. То же самое произойдет и с пунктами основного меню: их пометка будет строго соответствовать пунктам контекстного меню.

26. ПАНЕЛЬ ИНСТРУМЕНТОВ — ToolBar

С этой главы мы начнем разработку приложения, которое будет использоваться в нескольких последующих главах, поэтому будьте внимательны при создании примера, описанного в данной главе. В конечном итоге у нас получится простейший графический редактор, способный выполнять элементарные операции по созданию изображений.

ОБЩИЕ СВЕДЕНИЯ

На сегодняшний день вы вряд ли встретите приложение Windows, не имеющее панели инструментов. Как правило, кнопки на панели инструментов копируют команды некоторых разделов меню. На панель инструментов выносятся наиболее часто используемые кнопки, для того чтобы не тратить время на выбор этой команды из пункта меню. Панелей инструментов в одном окне может быть несколько, они могут располагаться в любой части окна, их можно передвигать и скрывать. В данной главе мы рассмотрим некоторые возможности применения панели инструментов в приложениях на C#.

Создайте новое Windows Forms приложение с именем GraphEditorApp. Измените свойство Text формы на «Графический редактор». Начнем проектирование панели инструментов приложения. Панель инструментов представлена в C# классом ToolBar. Соответствующий элемент находится в панели ToolBox Visual Studio .NET. Панель инструментов может содержать в себе различные элементы управления: кнопки, списки и даже диалоги. Мы ограничимся рассмотрением простейшего случая — использование кнопок в панели инструментов. Основной характеристикой кнопок является их изображение. Как правило, кнопка в панели инструментов хранит пиктографическое изображение того действия, которое совершается по нажатию кнопки. Для того чтобы создать ToolBar, вам необходимо предварительно подготовить изображения для его кнопок.

РАБОТА С РЕДАКТОРОМ ИЗОБРАЖЕНИЙ

Я долго не мог решить для себя проблему, каким образом создать пример приложения с использованием изображений, которые присутствовали бы у каждого из вас. Я не мог найти ни в стандартной поставке Windows, ни в Visual Studio .NET необходимых для примера пиктографических изображений. Я бы мог выложить изображения в Интернет, но не

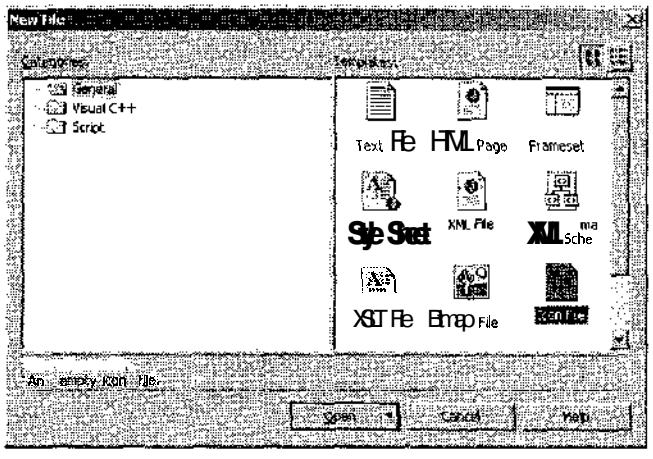
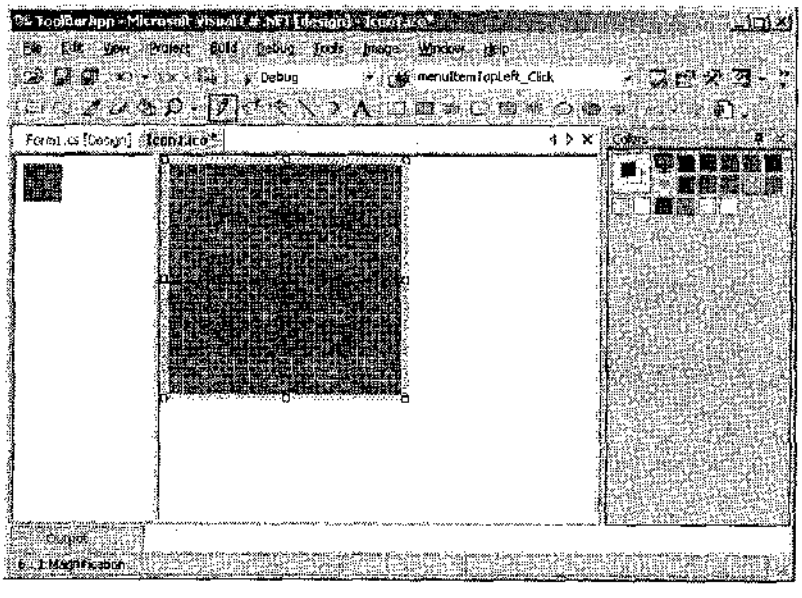


Рис. 26.1. Диалог выбора типа файла

у каждого может быть доступ к Сети. Поэтому вывод только один — вам придется самостоятельно создать изображения. Я не буду описывать, как нужно водить курсор мыши, для того чтобы это сделать. Я лишь поясню, как в Visual Studio.Net создавать собственные изображения. Для работы с изображениями в среде разработки существует Image Editor. Он позволяет создать изображения с использованием простейших инструментов. Для создания файла с изображением вам необходимо воспользоваться пунктом меню *File/New/File...* В появившемся окне *New File* (рис. 26.1) выберите тип файла «Icon File».

Перед вами появится пустое изображение с дополнительной панелью управления. Кроме того, в основном меню появится новый пункт меню *Image*. Выберите в меню пункт *Image/Show Colors Window*. На экране вы увидите панель с отображением палитры компонент. В целом, среда Visual Studio .NET должна выглядеть так, как показано на рис. 26.2.

Рис 26.2. Редактор изображений Visual Studio .NET



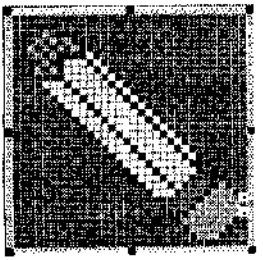


Рис. 26.3. Пиктограмма инструмента «Карандаш»

Я не стану описывать, как использовать компоненты для рисования, ведь все, что вы нарисуете — будете использовать только вы. Поэтому попробуйте создать приемлемое для себя изображение элемента «Карандаш». То, что получилось у меня, изображено на рис. 26.3.

По умолчанию Visual Studio .NET создает пиктограмму с двумя изображениями 16x16 пикселов и 32x32 пикселов. Вы рисовали изображение размером 32x32. Для того чтобы избежать неприятностей при дальнейшем использовании пиктограммы, необходимо удалить изображение 16x16 из файла. Для этого выберите из меню *Image/Current Icon Image Types/16×16*. В рабочей области появится новое изображение. Удалите его, используя меню *Image/Delete Image Type*.

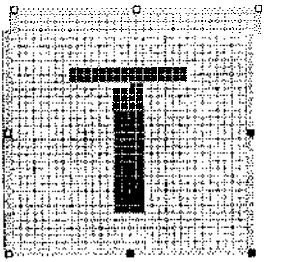


Рис. 26.4. Пиктограмма инструмента «Текст»

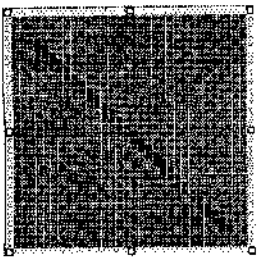


Рис. 26.5. Пиктограмма инструмента «Линия»

Сохраните изображение в файл с именем «Карандаш.ico». Кроме элемента «Карандаш» вам необходимо создать пиктограммы для инструментов «Текст», «Линия» и «Эллипс». Примеры пиктограмм представлены на рис. 26.4, 26.5, 26.6.

Не забудьте удалить пиктограммы размером 16x16 и из этих файлов. Сохраните созданные изображения в файлы с именами «Текст.ico», «Линия.ico» и «Эллипс.ico», соответственно. Если вы справились с этим заданием, значит, можно приступать к дальнейшей разработке приложения.

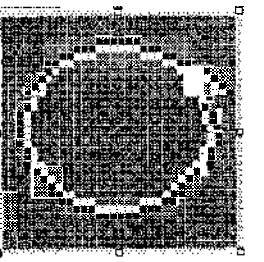


Рис. 26.6. Пиктограмма инструмента «Эллипс»

СОЗДАНИЕ ПАНЕЛИ ИНСТРУМЕНТОВ

Для начала необходимо выполнить подготовительную операцию — настроить список изображений для кнопок панели инструментов. Добавьте на форму элемент управления ImageList. Работа с ImageList уже описывалась в главе 23. Поместите в список изображений те четыре пиктограммы, которые создали в предыдущем разделе.

Добавьте на форму компонент ToolBar. На форме появится пустая панель инструментов (рис. 26.7).

Свойство ImageList добавленного объекта toolBarl измените на imageListl. Таким образом, вы свяжете созданный список изображений с панелью

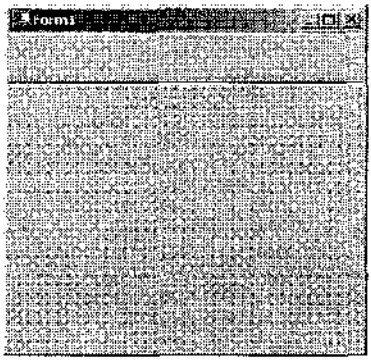


Рис. 26.7. Проект формы с пустой панелью инструментов

инструментов. Добавим к панели инструментов необходимые кнопки. Для этого в окне свойств объекта toolBarl необходимо настроить свойство Buttons. Нажмите на кнопку с тремя точками в поле Buttons. Откроется окно *ToolBarButton Collection Editor*. Добавьте четыре кнопки и назовите их toolBarButtonPen, toolBarButtonFill, tool BarButton Line, ToolBarButtonEllipse, изменив свойство Name каждого элемента в правом окне редактора. Свойство Style каждой кнопки установите в ToggleButton. Это позволит кнопкам удерживаться в нажатом состоянии. Нам необходимо это свойство, потому как нажатая кнопка будет характеризовать текущий режим работы редактора. Последним шагом в настройке кнопок панели инструментов является установка изображений для каждой конкретной кнопки. Какое именно изображение из списка будет соответствовать кнопке, задается свойством ImageIndex. Выберите для каждой кнопки изображение, соответствующее ее индексу:

0 — toolBarButtonPen — Карандаш
1 — toolBarButtonText — Текст
2 — toolBarButtonLine — Линия
3 — ToolBarButtonEllipse — Эллипс

Рис. 26.8. Панель инструментов графического редактора

В итоге у вас получится панель инструментов, соответствующая рис. 26.8.

Вы уже можете запустить программу и посмотреть результаты своей работы. Программа будет содержать только панель инструментов, которая не выполняет пока никаких функций.

Добавьте в программу перечисление (enum), которое будет содержать режимы работы программы (инструменты). Для этого выше описания класса Forml добавьте описание перечисления.

```
namespace GraphEditorApp
{
  public enum Tools
  {
    PEN = 1, TEXT, LINE, ELLIPSE, NONE = 0
  }
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1: System.Windows.Forms.Form
  {
  ...
  }
```

В классе Form1 создайте объект перечисления.

```
public class Form1: System.Windows.Forms.Form
{
  public Tools currentTool;

}
```

Этот объект будет содержать выбранный в панели инструментов инструмент для рисования. Впоследствии программа будет использовать эту переменную для определения того, какой инструмент выбран текущим.

Для того чтобы завершить работу с панелью инструментов приложения, добавьте обработчик события ButtonClick объекта toolBarl, щелкнув два раза указателем мыши по имени события ButtonClick на закладке событий в окне свойств. При этом в программу добавится функция toolBar1_ButtonClick как обработчик события, происходящего при нажатии кнопки на панели инструментов. У вас может возникнуть вопрос: «Как мы узнаем о том, какая именно кнопка была нажата?» Да, действительно, мы не создаем отдельный обработчик для каждой кнопки панели управления. Но мы можем различить это по параметрам, передаваемым в функцию. Измените метод toolBar1_ButtonClick и добавьте дополнительный код, как показано ниже:

```
private void toolBar1_ButtonClick(object sender,
System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
  switch(e.Button.ImageIndex)
  {
    case 0:
      // установка режима рисования карандашом
      currentTool = Tools.PEN;
      break;
    case 1:
      // установка режима текста
      currentTool = Tools.TEXT;
      break;
    case 2:
      // установка режима рисования линий
      currentTool = Tools.LINE;
      break;
    case 3:
      // установка режима рисования эллипсов
      CurrentTooi. = Tools.ELLIPSE;
      break;
  }

  SetToolBarButtonsPushedState(e.Button);
}
private void SetToolBarButtonsPushedState(ToolBarButton button)
```

```
{
  foreach(ToolBarButton btnItem in toolBar1.Buttons)
  {
    if(btnItem == button)
    {
      btnItem.Pushed = true;
    }
    else
    {
      btnItem.Pushed = false;
    }
  }
}
```

Обработчик нажатия кнопок на панели инструментов — функция toolBar1_ButtonClick — включает в себя инструкцию switch, которая позволяет определить, какая именно кнопка была нажата. Вторым параметром функции-обработчика является переменная типа ToolBarButtonClickEventArgs. Класс ToolBarButtonClickEventArgs имеет свойство Button, которое соответствует нажатой кнопке. Если свойство ImageIndex кнопки совпадает с искомым индексом в инструкции case, то будет выполняться соответствующая операция. В соответствии с нажатой кнопкой выставляется режим работы программы (текущий выбранный инструмент).

Функция SetToolBarButtonsPushedState предназначена для контроля состояния кнопок панели инструментов. Если вы запускали программу до изменения вышеприведенного кода, то при нажатии кнопок на панели инструментов они нажимались независимо друг от друга. Но для нашего приложения свойственно существование только одного режима в один момент времени. Поэтому может быть нажатой только одна из кнопок.

Функция SetToolBarButtonsPushedState следит за состоянием нажатия кнопок. При помощи цикла foreach функция просматривает все кнопки панели инструментов. Та из кнопок, которая совпадает с переданным параметром, делается вдавленной, свойство Pushed устанавливается в true. Для всех остальных кнопок свойство Pushed устанавливается в false.

Вызов функции SetToolBarButtonsPushedState происходит из обработчика нажатия кнопок toolBar1_ButtonClick с параметром, соответствующим нажатой кнопке. Поэтому нажатая кнопка будет вдавлена, а все остальные нет.

Теперь при запуске программы вы увидите, что только одна из кнопок может быть вдавлена в один момент времени. Как только вы нажмете Другую кнопку, предыдущая вдавленная примет свое обычное состояние.

На этом мы завершим разработку панели инструментов приложения, но не закончим разработку самого приложения. Нам еще много предстоит сделать, чтобы созданная нами панель инструментов выполняла назначенные ей функции. Сохраните приложение GraphEditorApp, используя пункт меню *SaveAll*.

27. СОЗДАНИЕ MDI ПРИЛОЖЕНИЙ

MDI (Multiple-document interface) приложения позволяют отображать сразу несколько документов одновременно. При этом каждый документ будет отображаться в своем собственном окне. Обычно MDI приложения имеют в основном меню подпункты для переключения между окнами и документами.

РОДИТЕЛЬСКИЕ И ДОЧЕРНИЕ ФОРМЫ

Механизм работы MDI приложений немного сложнее, чем обычных Windows Forms приложений, базирующихся на диалогах. Основным окном MDI приложения является родительская форма. Она может содержать несколько дочерних окон. Только одно из дочерних окон может быть активно в один момент времени.

Создание родительской формы

Мы не будем создавать новое приложение. Откройте проект программы GraphEditorApp, который был создан при изучении предыдущей главы.

Свойство IsMdiContainer установите в true, это будет определять форму как родительское окно MDI приложения.

Создание головного меню

Создайте основное меню программы (глава 25). Добавьте в меню пункты «&Файл», «&Инструмент» и «&Окно». Измените их свойство name на menuItemFile, menuItemTool и menuItemWindow соответственно.

Добавьте в меню «&Файл» пункт «&Создать». Свойство name установите в menuItemNew. Этот пункт меню будет предназначен для создания дочерних окон.

В меню «&Инструмент» добавьте пункты «&Карандаш», «&Текст», «&Линия», «&Эллипс». Свойство name для них установите как:

«&Карандаш» — menuItemPen;

«&Текст» — menuItemText;

«&Линия» — menuItemLine;

«&Эллипс» — menuItemEllipse.

Эти пункты меню будут предназначены для выбора режима работы программы, точно так же, как это делается из панели инструментов.

Пункт меню «&Окно» будет содержать список всех открытых дочерних окон. Такая возможность заложена в меню автоматически. Вам необходимо лишь выставить для пункта меню свойство MdiList в true. Установите свойство MdiList пункта меню «&Окно» в true.

Создание обработчиков меню

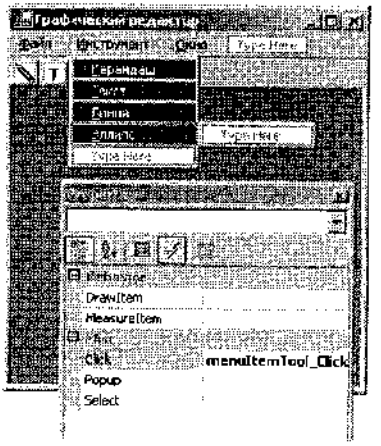


Рис. 27.1. Изменение имени обработчика события

Создайте один общий обработчик для подпунктов меню пункта «&Инструмент». Для этого выделите все четыре пункта меню, удерживая клавишу Ctrl, и щелкните два раза указателем мыши по событию Click в окне свойств. Замените имя события, созданное по умолчанию, на имя menuItemTool_Click. Для этого просто измените строку в поле, соответствующем имени события, в окне свойств (рис. 27.1).

Как вы уже, наверное, догадались, функция menuItemTool_Click будет выполнять ту же задачу, что и функция toolBar1_ButtonClick нашего приложения, то есть устанавливать инструмент для рисования при выборе определенного пункта меню. Кроме того, не стоит забывать о том, что выбранный инструмент должен быть помечен в меню. Поэтому мы также должны создать аналог функции SetToolBarButtonsPushedState для элементов меню. Найдите в коде программы функцию menuItemTool_Click и измените ее код так, как продемонстрировано ниже.

```
private void menuItemTool_Click(object sender, System.EventArgs e)
{
  // получаем пункт меню
  MenuItem item = (MenuItem)sender;

  switch(item.Text)
  {
    case "&Карандаш":
      // установка режима рисования карандашом
      currentTool = Tools.PEN;
      break;
    case "&Текст":
      // установка режима заливки
      currentTool = Tools.FILL;
      break;
    case "&Линия":
      // установка режима рисования линий
      currentTool = Tools.LINE;
      break;
```

```
    case "&Эллипс":
      // установка режима рисования эллипсов
      currentTool-••Tools.ELLIPSE;
      break;
  }

  // устанавливаем состояние пунктов меню
  SetMenuItemsCheckedState (item);

  // устанавливаем состояние кнопок
  SetToolBarButtonsPushedState(toolBar1.Buttons[item.Index]);
}

private void SetMenuItemsCheckedState (MenuItemitem)
{
  // для каждого пункта меню
  foreach(MenuItem menuItem in menuItemTool.MenuItems)
  i
    if(menuItem == item)
    {
      menuItem.Checked - true;
    }
    else
    {
      menuItem.Checked = false;
    }
  }
}
```

Кроме изменения кода обработчика, я добавил код функции для пометки пунктов меню. Функция SetMenuItemsCheckedState устанавливает флажок возле нужного пункта меню. Она принимает в качестве параметра пункт меню, который послал сообщение. Используя инструкцию foreach, перебираются все подпункты меню «Инструмент». Если подпункт совпадает с переданным параметром, то свойство Checked устанавливается в true, если не совпадает, то свойство Checked устанавливается в false.

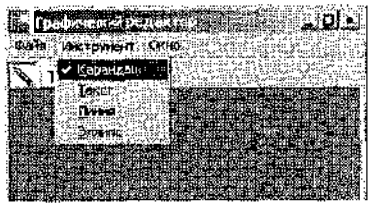
Для того чтобы кнопки на панели инструментов и пункты меню работали синхронно, необходимо дописать еще несколько строк кода. Как вы заметили, функция menuItemTool_Click вызывает метод для установки состояния кнопок:

```
SetToolBarButtonsPushedState (toolBar1.Buttons[item.Index]);
```

Таким образом, при выборе пункта меню соответствующая кнопка в панели инструментов перейдет в нажатое состояние. То же самое нужно сделать при обработке нажатия кнопок панели инструментов. Необходимо (кроме вызова функции SetToolBarButtonsPushedState, что уже сделано) вызвать функцию SetMenuItemsCheckedState. Добавьте вызов метода

SetMenuItemsCheckedState после вызова SetToolBarButtonsPushedState в функции toolBarl_ButtonClick.

```
SetToolBarButtonsPushedState(e.Button);
SetMenuItemsCheckedState (menuItemTool.MenuItems[e.Button.ImageIndex]);
```



Запустите программу. Теперь вы можете выбирать инструмент как из панели инструментов, так и из меню. При этом выбранный инструмент будет помечен и в меню, и в панели инструментов (рис. 27.2).

Рис. 27.2. Меню выбора инструмента

Создание дочерних окон

Шаблон дочернего окна

Любое окно, существующее в приложении, может быть дочерним. Разработчик сам создает шаблон формы, которая будет являться дочерним окном. Для того чтобы создать шаблон дочернего окна, добавьте в приложение новую форму. Для этого выберите из меню пункт *Project/Add Windows Form ....* Пометьте в правом списке пункт Windows Form и нажмите кнопку *Open.* Перед вами откроется окно дизайнера форм и новая форма с именем Form2. Измените некоторые свойства Form2:

Text — Новое окно

BackCoIor — Window.

Обработка событий

Создание дочерних окон должно происходить при выборе пункта меню «Создать». Для этого нам необходимо создать обработчик этого пункта меню. Щелкните два раза указателем мыши по имени события Click пункта меню «Создать» в окне свойств. В код программы добавится обработчик события Click с именем menuItemNew_Click. Добавьте к обработчику события menuItemNew_Click представленный ниже код:

```
private void menuItemNew_Click(object sender, System.EventArgs e)
{
  Form2 newMDIChild = new Form2();
  newMDIChild.MdiParent = this;
  newMDIChild.Show();
}
```

В функции создается экземпляр класса Form2 с именем newMDIChild. Объект newMDIChild — это обычное окно. Для того чтобы создаваемое окно отображалось как дочерняя форма приложения, необходимо установить его свойство MdiParent равным this. Таким образом, мы укажем, что родительской

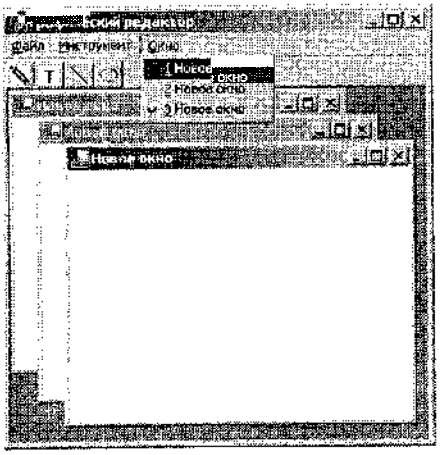


Рис. 27.3. Список открытых дочерних окон

формой создаваемого окна является главная форма приложения. Все, что остается сделать, это отобразить форму на экране. Для этого используется метод Show.

Запустите программу. Выберите из меню пункт *Файл/Создать*. На экране появится дочернее окно с именем «Новое окно». Вы можете создать еще несколько дочерних окон, воспользовавшись все тем же пунктом меню. При этом пункт меню *Окно* будет содержать список всех дочерних окон, открытых на текущий момент (рис. 27.3).

Сохраните приложение GraphEditor. Оно еще пригодится в обсуждении следующих глав.

28. ОБРАБОТКА СООБЩЕНИЙ МЫШИ

Мышь стала неотъемлемым атрибутом при работе в Windows. Поэтому в любой программе вы должны предоставлять пользователю возможность выполнить любое действие при помощи мыши. Исключение составляет ввод символов с клавиатуры.

ВИДЫ СОБЫТИЙ

Для обработки сообщений мыши в C# предусмотрен ряд событий, которые посылаются программе при совершении определенных действий. События посылаются, если вы передвинете курсор мыши, щелкните какой-нибудь кнопкой либо проделаете все эти действия одновременно.

Для обработки сообщений от мыши у формы существуют следующие события (рис. 28.1):

MouseDown — обработка нажатия какой-либо из кнопок вниз;

MouseEnter — вызывается при попадании указателя мыши в область формы;

MouseHover — вызывается при зависании указателя мыши в окне формы;

MouseLeave — вызывается при покидании курсора мыши области формы;

MouseMove — вызывается при движении мыши в области формы;

MouseUp — вызывается при отпускании кнопки мыши.

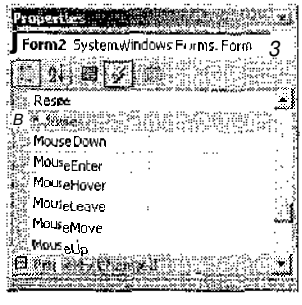


РИС. 28.1. Mouse events

Для примера программы обработки сообщений мыши будем использовать приложение GraphEditorApp. Откройте проект GraphEditorApp, с которым мы работали в предыдущей главе. Откройте в окне дизайнера форму Form2.

ПАРАМЕТРЫ СОБЫТИЙ

Создайте для формы Form2 обработчик события MouseDown. Для этого щелкните два раза указателем мыши по событию MouseDown в окне свойств. В коде программы появится функция-обработчик Form2_MouseDown. Реализуйте ее так, как показано в нижеприведенном коде:

```
private void Form2_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
```

```
{
  string text;
  MouseButtons button;
  button = e.Button;
  if (button == MouseButtons.Left)
  (
    text = "левую";
  }
  else if(button == MouseButtons.Right)
  i
    text = "правую";
  }
  else
  {
    text = "среднюю";
  }
    str_ng message = "Вы нажали " + text + " кнопку мыши в координатах:\n"+
  "x:= " + e.X.ToString() + "\n" +
  "y:= " + e.Y.ToString();
    MessageBox.Show(message);
}
```

Параметр функции MouseEventArgs содержит всю информацию о кнопке, пославшей сообщение. Свойство MouseEventArgs.Button хранит информацию о типе кнопки (левая, правая, средняя). Блок инструкций if ... else выбирает из трех возможных вариантов ту кнопку, которая передалась в качестве параметра. На основании полученной информации о кнопке формируется текст сообщения. Например, если вы нажмете левую кнопку мыши, то сообщение будет содержать строку «Вы нажали левую кнопку мыши». В конец строки сообщения дописываются координаты окна, в которых был совершен щелчок указателем мыши.

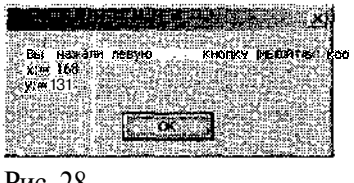


Рис. 28.2. Сообщение о нажатии левой кнопки мыши

Запустите программу. Создайте новое окно, используя меню *Файл/Создать*. Щелкните курсором в любом месте дочернего окна. На экране появится сообщение, аналогичное тому, которое изображено на рис. 28.2.

Точно так же можно обрабатывать другие сообщения от мыши. Теперь сохраните приложение GraphEditorApp. Этот пример еще пригодится в следующей главе.

29. РАБОТА С ГРАФИКОЙ

ОСОБЕННОСТИ GDI+

Для рисования объектов в Windows Forms приложениях язык C# содержит очень богатый набор методов. Пространство имен Drawing содержит множество объектов, которые облегчают программисту работу с графикой. Специально для .NET платформы разработчики Microsoft разработали GDI+ библиотеку, значительно повысив возможности GDI (Graphic Device Interface). GDI+ включает возможности рисования простейших объектов (линии, эллипсы...), рисование различных объектов 2D графики, отображение файлов различных графических форматов (bmp, jpeg, gif, wmf, ico, tiff...) и многое другое. Мы рассмотрим работу с простейшими объектами для рисования: линиями, окружностями, шрифтами.

РИСОВАНИЕ ОБЪЕКТОВ

Откройте приложение GraphEditorApp, с которым мы работали в предыдущих главах. У нас уже реализованы меню, панель инструментов, создание дочерних окон, обработка сообщений от мыши. Теперь мы узнаем о том, как рисовать графические примитивы. Для начала следует уточнить функциональность программы.

Для рисования линии вам необходимо будет два раза щелкнуть мышью в различных частях окна. При этом линия нарисуется от координаты, в которой произошел первый щелчок мышью в координату, где произошел второй щелчок мышью. Это нетипично для графического редактора. Обычно линия тянется от координаты первого щелчка мыши за курсором до тех пор, пока пользователь не отпустит кнопку. Но такая функциональность потребует усложнить код программы. Для наших учебных целей мы обойдемся двумя щелчками указателя мыши.

Рисование эллипсов аналогично рисованию линий. Первый щелчок мыши будет задавать левую верхнюю координату эллипса, а второй — правую нижнюю координату.

Рисование текста. При щелчке мыши строка текста будет отображаться справа от курсора.

Особый интерес представляет рисование карандашом. Оно должно происходить во время движения указателя мыши по области окна с нажатой левой кнопкой мыши. Для отображения пути движения указателя мыши мы будем рисовать линии между предпоследней координатой

курсора и последней. Для этого нам придется постоянно запоминать координату последнего положения курсора мыши.

Ну что же, теоретическая часть на этом завершена, приступим к примеру.

Рисование карандашом

Откройте из окна SolutionExplorer форму Form2. Эта форма описывает дочернее окно нашего приложения. Именно в классе Form2 нам необходимо написать код, который позволит рисовать графические примитивы.

У нас в программе уже присутствует обработчик события MouseDown. Вам необходимо добавить обработчики для событий MouseMove и MouseUp. Сделайте это, щелкнув два раза по соответствующим полям на закладке событий в окне свойств. Тот код, который у вас присутствовал в обработчике события MouseDown, нам больше не понадобится. Вы можете удалить его либо закомментировать.

Измените код обработчиков событий так, как представлено ниже:

```
private void Form2_MouseDown(object sender,
  System.Windows.Forms.MouseEventArgs e)
{
  Form1 parentForm = (Form1)MdiParent;

  switch(parentForm.currentTool)
  {
    case Tools.LINE:
      //DrawLine(new Point (e.X, e.Y));
      break;
    case Tools.ELLIPSE:
      //DrawEllipse(new Point (e.X, e.Y));
      break;
    case Tools.TEXT:
      //DrawText(new Point(e.X, e.Y));
      break;
    case Tools.PEN:
      // устанавливаем флаг для начала рисования карандашом
      drawPen = true;
      break;
  }

  // запоминаем первую точку для рисования
  PreviousPoint.X = e.X;
  PreviousPoint.Y = e.Y;
}

private void Form2_MouseUp(object sender,
  System.Windows.Forms.MouseEventArgs e)
```

```
{
  drawPen = false;
}
private void Form2_MouseMove(object sender,
  System.Windows.Forms.MouseEventArgs e)
{
  // если курсов еще не отпущен
  if(drawPen)
  {
    // создаем объект Pen
    Pen blackPen = new Pen(Color.Black, 3);

    // получаем текущее положение курсора
    Point point = new Point(e.X, e.Y);
      // создаем объект Graphics
    Graphics g = this.CreateGraphics();

    // рисуем линию
    g.DrawLine(blackPen, PreviousPoint, point);
      // сохраняем текущую позицию курсора
    PreviousPoint = point;
  }
}
```

Когда вы введете этот код, программа не будет компилироваться. Дело в том, что я добавил в класс Form2 новые поля и методы. Вы должны были это заметить. В коде программы используются необъявленные пока переменные PreviousPoint, drawPen и необъявленные методы DrawLine, DrawEllipse и Text. Последние закомментированы и не мешают пока компиляции программы. А вот переменные необходимо объявить. Давайте объявим в программе переменные PreviousPoint и drawPen:

```
public class Form2: System.Windows.Forms.Form
{
    public bool drawPen;
    public Point PreviousPoint;
}
```

Переменные следует объявить членами класса Form2. drawPen — это двоичная переменная, которая будет определять, рисовать карандашу или нет. Обработчик Form2_MouseDown выставляет значение переменной drawPen в true, если в момент нажатия кнопки мыши установлен инструмент «Карандаш». Обработчик Form2_MouseUp выставляет значение переменной назад в false. Таким образом, при нажатии кнопки мыши флаг drawPen устанавливается в true и не сбрасывается в false до тех пор, пока пользователь не отпустит кнопку вверх. Обработчик Form2_MouseMove

рисует на экране линии только тогда, когда переменная drawPen находится в true. Рисование происходит следующим образом:

1. Создается объект Pen, который инициализируется черным цветом (Color.Black) и толщиной линии, равной 3.

```
Pen blackPen = new Pen(Color.Black, 3);
```

2. Получаются текущие координаты курсора мыши.

```
Point point = new Point(e.X, e.Y);
```

3. Создается объект Graphics на базе текущей формы.

```
Graphics g = this.CreateGraphics();
```

4. Вызывается метод DrawLine объекта Graphics. В качестве координат линии передаются предыдущая и текущая координаты курсора мыши. После прорисовки линии текущая координата курсора мыши запоминается в переменную, которая хранит предыдущую координату.

```
PreviousPoint = point;
```

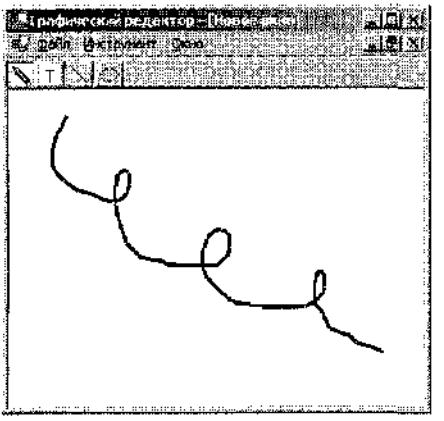


Рис. 29.1. Рисование карандашом

Таким образом, каждое движение мыши с нажатой кнопкой будет заставлять программу рисовать линию между двумя соседними точками. В итоге получится кривая пути, по которому двигался курсор мыши.

Запустите приложение. Создайте новое окно. Выберите режим рисования карандашом. Попробуйте нарисовать в окне любую кривую линию, удерживая курсор мыши в нажатом состоянии (рис. 29.1). Если у вас получилось нарисовать кривую — вы все сделали правильно, если нет — внимательно перечитайте приведенное в книге описание. Если и потом у вас все же что-то не получилось — не расстраивайтесь. Я понимаю, что приведенный в этой главе пример довольно громоздкий, поэтому вынес полный листинг кода программы GraphEditorApp в Приложение. Вы можете набрать программу заново из приложения либо сравнить мой код со своим на предмет наличия ошибок.

Однако наша программа еще не завершена. Нам предстоит реализовать функции рисования линий, эллипсов и написания текста.

Рисование текста и графических примитивов

Для этого добавьте в класс Form2 описание приведенных ниже функций:

```
void DrawLine(Point point)
{
  // если один раз уже щелкнули
  if (FirstClick == true)
```

```
  {
    // создаем объект Pen
    Pen ЧаскРеп = new Pen(Color.Black, 3);

    // создаем объект Graphics
    Graphics g = this.CreateGraphics();

    // рисуем линию
    g.DrawLine(ЧаскРеп, PreviousPoint, point);

    FirstCIick = false;
  }
  else
  {
    FirstCIick = true;
  }
}

void DrawEllipse(Point point]
[
  // если один раз уже щелкнули
  if (FirstCIick == true)
  {
    // создаем объект Pen
    Pen ЧаскРеп = new Pen(Color.Black, 3);

    // создаем объект Graphics
    Graphics g = this.CreateGraphics();

    // рисуем эллипс
    g.DrawEllipse( ЧаскРеп,
      PreviousPoint.X, PreviousPoint.Y,
      point.X - PreviousPoint.X, point.Y - PreviousPoint.Y);

    FirstClick = false;
  }
  else
  {
    FirstCIick = true;
  }
}

void DrawText(Point point)
t
  // создаем объект Graphics
  Graphics g = this.CreateGraphics();
```

```
    // создаем объект Font
  Font titleFont = new Font("Lucida Sans Unicode", 15);
    // рисуем текст красным цветом
  g.DrawString("Программирование на C#",
    titleFont, new SolidBrush(Color.Red), point.X, point.Y);
}
```

Методы DrawLine и DrawEllipse используют переменную FirstClick. Добавьте объявление этой переменной в класс Form2.

```
public bool drawPen;
public bool FirstClick;
public Point PreviousPoint;
```

Методы DrawLine и DrawEllipse рисуют объекты по двум координатам точек на экране. При первом щелчке мыши запоминается первая координата и выставляется значение переменной FirstClick в true. При втором щелчке мыши происходит рисование линии, и значение FirstClick сбрасывается в false.

Рис. 29.2. Использование приложения «Графический редактор»

Для рисования текста используется метод DrawText. Для прорисовки сначала создается объект Font со шрифтом типа Lucida Sans Unicode размером 15 единиц. Затем при помощи метода DrawString строка текста «Программирование на C#» выводится на экран.

Теперь вы можете откомментировать вызовы методов DrawLine, DrawEllipse и DrawText в функции Form2_MouseDown.

Запустите программу. Выбирая различные режимы работы программы, вы можете создавать простейшие графические изображения (рис. 29.2).

30. РАБОТА С КЛАВИАТУРОЙ

Мы уже рассмотрели в главе 28 обработку сообщений мыши. Однако как бы не было полезно использование мыши, без клавиатуры не обойтись. Скажу больше: все коммерческие приложения должны иметь возможность выполнить любую команду, как при помощи мыши, так и при помощи клавиатуры.

СООБЩЕНИЯ КЛАВИАТУРЫ

Для обработки сообщений с клавиатуры в Windows Forms приложениях предусмотрены три события: KeyUp, Keypress, KeyDown.

Событие KeyUp посылается при отпускании кнопки на клавиатуре.

Событие KeyPress посылается первый раз при нажатии кнопки на клавиатуре вместе с событием KeyDown и затем может посылаться неограниченное число раз, если пользователь удерживает клавишу в нажатом состоянии. Частота посылки события KeyPress зависит от настроек операционной системы. Событие KeyDown посылается при нажатии кнопки на клавиатуре.

Для примера создадим приложение, которое будет обрабатывать нажатие клавиш и выводить на экран информацию о том, какая клавиша была нажата. Для этого создайте новое Windows Forms приложение с именем KeyboardApp. Измените свойства вновь созданной формы:

- Text — «Информация о нажатых клавишах»;
- KeyPreview — True.

Свойство Text задает заголовок окна. Свойство KeyPreview определяет свойство, позволяющее форме улавливать сообщения клавиатуры от дочерних элементов управления формы. Если свойство KeyPreview формы устанавлено в False, то форма не будет получать сообщения от клавиатуры, если активизирован один из элементов формы. Другими словами, если на форме присутствует компонент TextBox и курсор мыши находится в его поле, то при нажатии клавиши на клавиатуре форма об этом узнать не сможет. Поэтому, если вы собираетесь обрабатывать клавиатуру в классе формы, вам необходимо выставлять свойство KeyPreview в true.

Добавьте на форму элемент управления TextBox.

Измените некоторые свойства элемента TextBox:

- Text — «»;
- ReadOnly — True;
- TabStop — False.

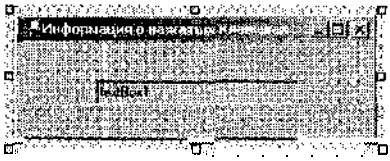


Рис. 30.1. Проектирование формы приложения KeyboardApp

Форма должна выглядеть так, как показано на рис. 30.1.

Добавьте в программу обработчик события KeyDown. Для этого в окне свойств формы щелкните два раза указателем мыши по событию KeyDown в окне свойств формы. В код программы включится обработчик события KeyDown с именем Form1_KeyDown. Добавьте к обработчику код, представленный ниже:

```
private void Form1_KeyDown(object sender, System.Windows.Forms.KeyEventArgs e)
{
  // очищаем поле
  textBox1.Text = "";

  // проверяем нажата ли клавиша Ctrl
  // если да, то записаваем в поле слово Ctrl
  if(e.Control)
   {
    textBox1.Text += "Ctrl+";
   }

  // проверяем нажата ли клавиша Shift
  // если да, то записаваем в поле слово Shift
  if (e.Shift)
   {
    textBox1 .Text += "Shift+";
   )
  // проверяем нажата ли клавиша Alt
  // если да, то записаваем в поле слово Alt
  if (e.Alt)
   {
    textBox1.Text += "Alt+";
   }

  // копируем KeyData нажатой клавиши
  Keys key = e .KeyData;

  // извлекаем из данных о нажатой клавише
  // коды системных кнопок, таких как
  // Ctrl, Shift, Alt
  key &= ~Keys.Control;
  key &= ~Keys.Shift;
  key &= ~Keys.Alt;

  // выводим полученное словосочетание
  textBox1.Text += key.ToString();
}
```

КЛАСС **KeyEventArgs**

Класс KeyEventArgs содержит всю информацию о нажатой клавише. Вот свойства, которые обычно используются при обработке нажатия кнопки:

- Alt — True, если нажата клавиша **Alt;**
- Control — True, если нажата клавиша **Ctrl;**
- Shift — True, если нажата клавиша **Shift;**
- KeyCode — код нажатой клавиши;
- KeyData — совокупность кодов нажатых клавиш;
- KeyValue — десятичное значение свойства KeyData;
- Handled — флаг, указывающий, было ли сообщение обработано. По умолчанию, значение Handled равно false. Если вы не хотите дальнейшей обработки нажатия кнопки, выставьте флаг Handled в true.

Для вывода на экран информации о нажатых управляющих клавишах мы проверяем значения свойств **Alt, Ctrl** и **Shift.** Если одна из клавиш нажата, то в поле TextBox добавляется соответствующее слово.

Для того чтобы вывести на экран информацию об основной нажатой клавише, мы должны предварительно удалить из поля KeyData системную информацию. Поэтому мы сбрасываем в 0 флаги управляющих клавиш свойства KeyData[1].

Рис. 30.2. Информация о нажатых клавишах

Инструкция key & = ~Keys.Control позволяет удалить из переменной key код управляющей клавиши **Ctrl** независимо от того, есть он там или нет. Точно так же удаляются коды клавиш **Alt** и **Shift.**

Метод ToString возвращает словесное описание кода клавиши. КОД обычНОЙ КЛаВИШИ ВЫводится одной буквой; код системной клавиши выводится словом, соответствующим клавише.

Запустите программу. Попробуйте нажимать кнопки на клавиатуре, сочетая их с управляющими клавишами. На экране будет отображаться информация в виде строки, содержащей все нажатые клавиши (рис. 30.2).

[1] Управляющие клавиши представлены битовыми значениями соответствующего разряда. В общем случае, свойство KeyData состоит из объединения логическим знаком ИЛИ кодов управляющих клавиш и кода основной клавиши. Если вы хотите оставить в KeyData только код основной клавиши, то вам необходимо удалить информацию о системных клавишах.

31. ТАЙМЕР И ВРЕМЯ

Очень часто приходится устанавливать зависимость выполнения каких-либо действий в приложении от времени. Это может быть выполнение периодически повторяющихся действий либо срабатывание команды в определенный момент времени.

КОМПОНЕНТ Timer

Работа с таймером в Windows Forms приложениях основана на все том же механизме событий. Вы устанавливаете таймер на определенную частоту, и операционная система будет рассылать вашему приложению события оповещения с указанной частотой.

Компонент Timer позволяет легко и просто работать со временем. Основными свойствами компонента Timer являются:

- Interval — задает период приема сообщений таймером в миллисекундах;
- Enabled — определяет состояние Включения/Выключения таймера;

Для работы с таймером вам необходимо лишь поместить на форму компонент Timer, установить его свойство Interval на заданный интервал времени и обработать событие Elapsed.

КОМПОНЕНТ DateTimePicker

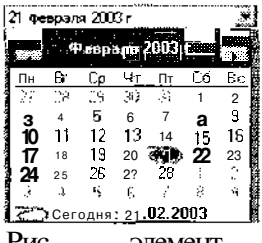


Рис. 31.1 элемент управления DateTimePicker

Элемент DataTimePicker представляет собой универсальный визуальный компонент для представления информации о времени. Он содержит компонент календарь (рис. 31.1) и позволяет легко изменять время в поле компонента.

Компонент DateTimePicker позволяет делать тонкую настройку формата отображения времени. Это достигается за счет возможности задания собственного формата отображения.

Основные свойства компонента DateTimePicker следующие:

- Format— позволяет установить один из стандартных форматов отображения времени либо указать свой;
- ShowUpDown — устанавливает тип элемента с правой стороны поля отображения. Если установить False —отображается ComboBox, открываю-

щий календарь, если True — отображается NumbericUpDown, изменяющий активное поле отображения;
- CustomFormat — строка, описывающая собственный формат отображения времени;
- MaxDate — максимально возможное время для ввода;
- MinDate — минимально возможное время для ввода;
- Value — значение времени.

СТРУКТУРА DateTime

Не знаю почему, но DateTime объявлена в C# как структура данных, а не класс, хотя имеет в своем составе и методы, и поля, и свойства. Однако вас это мало коснется. Как вы знаете, классы и структуры в C# очень похожи.

Структура DateTime предназначена для хранения и обработки переменных в формате даты или времени. Структура DateTime настолько универсальна, что ее используют и при работе со строками, и при работе с числами, и при работе с базами данных. Структура DateTime имеет отдельные свойства для каждой категории времени (год, месяц, число, час, минута, секунда, миллисекунда). Кроме того, DateTime имеет набор методов для обработки временных интервалов. Например, можно сложить два временных значения, вычесть, конвертировать в другой формат...

ФОРМАТ СТРОКИ ВРЕМЕНИ

Очень важной особенностью структуры DateTime является возможность перевода временного значения в строковый формат. При этом формат возвращаемой строки может быть задан динамически. Метод DateTime.ToString возвращает строку времени на основании формата аргумента. Вот некоторые константы, определяющие формат строки времени:
- dd — два знака дня месяца. Если день состоит из одной цифры — впереди ставится незначащий 0;
- dddd — день недели;
- MM — номер месяца (1—12);
- MMMM — название месяца;
- yyyy — номер года;
- hh — количество часов (1—12);
- HH — количество часов (1—24);
- mm — количество минут;
- ss — количество секунд.

Кроме того, строка формата может содержать любые разделительные символы для удобства представления. Вот несколько примеров:

Формат	Значение
dd MMMM yyyy HH:mm:ss	21 ноября 2002 14:48:56
dd.MM.yyyy HH:mm	21.11.2002 14:48
Сегодня dd MMMM yyyy года	Сегодня 21 ноября 2002 года

Как видно из таблицы, формат времени может быть очень удобно подстроен под конкретную задачу. И вам не придется самим склеивать строки.

Настройка формы

Для закрепления знаний о работе с таймером и временем напишем программу «Будильник», которая будет давать пользователю возможность выставить время срабатывания будильника, а по истечении этого времени на экран будет выводиться сообщение.

Создайте новое Windows Forms приложение с именем TimerApp. Измените свойство Text формы на «Будильник». Добавьте на форму два компонента DateTimePicker. Измените их свойства:

dateTimePicker1:
- Name — currentPicker;
- Format — Time;

dateTimePicker2:
- Name — timerPicker;
- Format — Time;
- ShowChekBox — True.

Один из элементов будет отображать текущее время, другой — время срабатывания таймера. Элемент timerPicker будет отображать CheckBox, определяющий, выставлен ли таймер в активное состояние.

Для улучшения восприятия интерфейса программы поместите элементы DateTimePicker в GroupBox элементы (рис. 31.2).

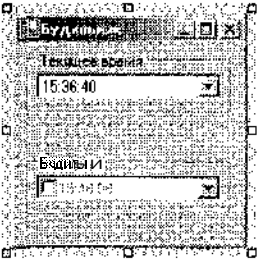


Рис. 31.2. Проектирование формы приложения «Будильник»

Обработка таймера

Поместите на форму приложения компонент Timer. Он находится в окне ToolBox на закладке Components. Мы оставим все свойства таймера по умолчанию. Далее необходимо создать обработчик единственного события Elapsed компонента Timer. Щелкните два раза по имени события Elapsed в окне свойств. При этом в коде программы создастся функция-обработчик с именем timer1_Elapsed. Измените код обработчика так, как показано ниже:

```
private void timer1_Elapsed(object sender,
    System.Timers.ElapsedEventArgs e)
```

```
{
  // получаем текущее значение времени
  DateTime currentTime = DateTime.Now;

  // обновляем значение элемента currentPicker
  currentPicker.Value = currentTime;

  // если будильник установлен
  if (timerPicker.Checked == true)
  {
  // сверяем текущее время с временем установленным
  // на будильнике с точностью до минуты
  if(timerPicker.Value.Minute == currentTime.Minute &&
     timerPicker.Value.Hour == currentTime.Hour &&
     timerPicker.Value.Second == currentTime.Second)
  {
    // отключаем будильник
    timerPicker.Checked = false;
      // выводим сообщение
    MessageBox.Show("Будильник активизирован");
  }
}
```

Таймер программы по умолчанию включен и принимает сообщения с частотой 10 раз в секунду. Это значит, что каждые 100 миллисекунд будет вызываться обработчик события Elapsed.

Для постоянного отображения времени в поле компонента currentPicker необходимо непрерывно изменять его значения, потому как по умолчанию он инициализируется текущим значением времени и больше не изменяется. Для изменения значения компонента мы получаем текущее время:

```
DateTime currentTime = DateTime.Now;
```

И затем присваиваем свойству Value компонента currentPicker это значение:

```
currentPicker.Value = currentTime;
```

Эти действия происходят каждые 100 миллисекунд. Если свойство Checked объекта timerPicker установлено в True, то проверяется совпадение текущего времени и времени, на которое установлен будильник. Проверяется только совпадение часов, минут и секунд, большей точности нам не надо. Если время совпадает, то свойство Checked выставляется в False и на экран выводится сообщение «Будильник активизирован».

32. ФАЙЛЫ

Пользователь должен иметь возможность сохранить результаты своей работы на диск и затем прочитать их. Иначе все наработки будут потеряны при выходе из приложения. Если даже вы не занимались ранее программированием, то должны знать, что вся информация на диске хранится в виде файлов. Файлы могут быть записаны в каталоги, а каталоги — вложены друг в друга. Язык C# предоставляет программистам возможность легко и просто сохранять и считывать данные с диска.

ПОНЯТИЕ ПОТОКОВ

В основе работы с файлами лежит понятие потоков. Поток ассоциируется с файлом и предоставляет набор методов для доступа к файлу через поток. Потоки обладают расширенными функциональными возможностями по сравнению с файлами. Потоки позволяют записывать и считывать структуры данных, массивы, другие потоки. Хотя поток и ассоциируется с файлом, не все данные из потока напрямую попадают в файл. Вся информация из потока заносится в буфер, и лишь при вызове определенных команд переносится в файл.

Основными классами для работы с файлами и потоками в C# являются File, FileStream и StreamReader. Класс File предназначен для создания, открытия, удаления, изменения атрибутов файла. Класс FileStream предназначен для чтения и записи информации в файл. Объекты этих классов работают в паре друг с другом. Механизм их взаимодействия очень прост и понятен.

Для работы с текстовыми файлами необходимо создать объект типа FileStream и проинициализировать его открытым файлом. Поскольку все методы класса File являются статическими (не привязаны к объектам), нет необходимости создавать экземпляр класса File. Вот типичный пример инициализации объекта FileStream:

```
FileStream myStream - File.Open("C:\MyFile.txt", FileMode.Open,
    FileAccess.Read);
```

В качестве дополнительного инструмента для работы с текстовыми файлами разработчиками C# были сделаны классы StreamReader и StreamWriter. Они позволяют читать и писать данные из потока построчно, посимвольно, сразу все. StreamReader и StreamWriter связываются с потоком при помощи конструктора инициализации:

```
StreamReader reader - new StreamReader (myStream);
StreamWriter writer = new StreamWriter(myStream);
```

АТРИБУТЫ ОТКРЫТИЯ ФАЙЛОВ

При открытии файла всегда необходимо указывать режим открытия файла и права доступа к файлу. В данном случае режим открытия установлен как FileMode.Open, что означает открыть файл, если он существует; права доступа установлены FileAccess.Re.ad, что означает возможность только читать файл. Функция Open возвращает объект типа FileStream, посредством которого в дальнейшем происходят чтение или запись в файл.

ДИАЛОГИ ОТКРЫТИЯ И СОХРАНЕНИЯ ФАЙЛОВ

Вы все хорошо знаете диалоги, предлагающие выбрать путь на диске и имя файла для открытия или записи (рис. 32.1).

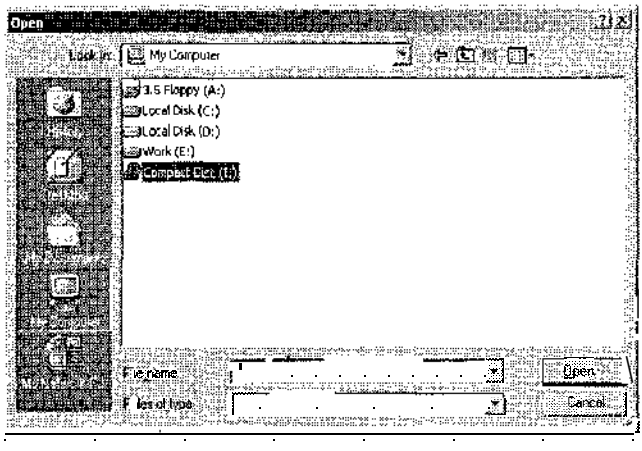


Рис. 32.1. Диалог открытия файла

С левой стороны диалога находятся кнопки быстрого выбора категории дискового пространства: «History», «Desktop», «My Documents», «My Computer», «Network». Кроме того, вы можете выбрать путь на диске, имя файла, расширение.

Для работы с диалогами открытия и сохранения файлов используются компоненты OpenFileDialog и SaveFileDialog. Они во многом схожи. Для отображения диалога открытия файла вам необходимо лишь создать объект класса OpenFileDialog и вызвать его метод ShowDialog. После закрытия диалога свойство FileName хранит имя выбранного файла и полный путь к нему.

В этой главе мы создадим простейший текстовый редактор, позволяющий читать текстовые файлы, редактировать информацию и сохранять ее в файл.

Создайте новое приложение с именем FileApp. Переименуйте свойство Text формы в «Текстовый редактор». Поместите на форму компонент TextBox и измените его свойства:

- Text — «»;
- Multiline — True;
- Dock — Fill.

При этом TextBox должен растянуться на весь экран. Свойство Multiline позволяет элементу вводить текст в несколько строк, а свойство Dock определяет положение элемента на форме. Если свойство Dock установлено в Fill, то элемент займет всю площадь формы.

Поместите на форму компонент MainMenu. Создайте в нем один пункт «Файл» с двумя подпунктами «Открыть» и «Сохранить». Измените свойство Name пунктов «Открыть» и «Сохранить» на menuItemOpen и menuItemSave. Создайте обработчики для пунктов меню «Открыть» и «Сохранить». Оставьте для них имена по умолчанию, просто щелкнув два раза указателем мыши по соответствующим пунктам меню. При этом в код программы должны были добавиться методы menuItemOpen_Click и menuItemSave_Click.

Добавьте на форму компоненты OpenFileDialog и SaveFileDialog. Для обоих установите свойство Filter как «Text files (*.txt)|*.txt». Это означает, что в диалоге будут показываться только файлы с расширением «txt».

Измените обработчики открытия и сохранения файлов так, как показано ниже:

```
private void menuItemOpen_Click(object sender, System.EventArgs e)
{
  // показываем диалог выбора файла
  openFileDialog1.ShowDialog();

  // получаем имя файла
  string fileName = openFileDialog1.FileName;

  // открываем файл для чтения и ассоциируем с ним поток
  FileStream stream = File.Open(fileName, FileMode.Open, FileAccess.Read);

  // если файл открыт
  if(stream != null)
  {
    // создаем объект StreamReader и ассоциируем
    // его с открытым потоком
    StreamReader reader = new StreamReader(stream);

    // читаем весь файл и записываем в TextBox
    textBox1.Text = reader.ReadToEnd();

    // закрываем файл
    stream.Close();
  }
}

private void menuItemSave_Click (object sender, System.EventArgs e)
{
  // показываем диалог выбора файла
  saveFileDialog1.ShowDialog();
```

```
        // получаем имя файла
        string fileName = saveFileDialog1.FileName;

        // открываем файл для записи и ассоциируем с ним поток
        FileStream stream = File.Open(fileName, FileMode.Create, FileAccess.Write);

        // если файл открыт
        if(stream != null)
        {
          // создаем объект StreamWriter и ассоциируем
          // его с открытым потоком
          StreamWriter writer = new StreamWriter(stream);

          // записываем данные в поток
          writer.Write(textBox1.Text);

          // переносим данные из потока в файл
          writer.Flush();

          // закрываем файл
          stream.Close();
        }
    }
```

Подробное описание работы методов дано в комментариях. Работа с чтением файла идет в 6 этапов:

- открытие файла;
- ассоциация файла с потоком;
- ассоциация потока со StreamReader;
- чтение данных;
- перенос данных в TextBox;
- закрытие файла.

Запись файла также проходит в 6 этапов:

- открытие файла;
- ассоциация файла с потоком;
- ассоциация потока со StreamWriter;
- запись данных;
- освобождение потока;
- закрытие файла.

Запустите приложение. Выберите пункт *Открыть*. В открывшемся диалоге (рис. 32.1) выберите текстовый файл. После нажатия кнопки *OK* данные из файла отобразятся в окне программы. Измените текст файла. Нажмите меню *Сохранить*. В открывшемся окне выберите новое имя файла, чтобы не затереть старый файл. После нажатия кнопки *OK* данные из программы перенесутся в файл. Вы можете убедиться в этом, воспользовавшись программой «Блокнот».

33. РАБОТА С БАЗАМИ ДАННЫХ

РЕЛЯЦИОННАЯ МОДЕЛЬ БАЗ ДАННЫХ

Для начала вам следует понять основополагающие моменты при проектировании реляционных баз данных. На сегодняшний день уже существует ряд СУБД, которые выполняют за вас большую часть работы при проектировании баз данных, например Microsoft Access, SQL server, Oracle, DB2, InterBase и т. д. Все они позволяют организовать доступ к своим ресурсам через язык SQL. SQL символизирует собой Структурированный Язык Запросов. Это язык, который дает возможность создавать и работать в реляционных базах данных, являющихся наборами связанной информации, сохраняемой в таблицах.

Что такое реляционная база данных?

Реляционная база данных — это тело связанной информации, сохраняемой в двумерных таблицах. Напоминает адресную или телефонную книгу. В книге имеется большое количество входов, каждый из которых соответствует определенной особенности. Для каждой такой особенности может быть несколько независимых фрагментов данных, например имя, телефонный номер и адрес.

Предположим, что вы должны сформатировать адресную книгу в виде таблицы со строками и столбцами. Каждая строка (называемая также записью) будет соответствовать определенной особенности; каждый столбец будет содержать значение для каждого типа данных — имени, телефонного номера и адреса, представляемого в каждой строке. Адресная книга могла бы выглядеть следующим образом:

Имя	Телефон	Адрес
Иван Иванов	234-23-12	ул. Московская, 12/237
Петр Петров	234-56-76	ул. Советская. 42/34
Олег Волков	23241-43	ул. Пушкинская, 2/17

То, что вы получили, является основой реляционной базы данных, двумерной (строка и столбец) таблицей информации. Однако реляционные базы данных редко состоят из одной таблицы. Такая таблица меньше, чем файловая система. Создав несколько таблиц взаимосвязанной информации, вы сможете выполнить более сложные и мощные операции с данными. Мощность базы данных зависит от связи, которую вы можете создать между фрагментами информации, а не от самого фрагмента информации.

Рассмотрим связь между данными на основе базы данных «Students», которую мы создадим. Это будет довольно простой пример связывания данных, хранящихся в разных таблицах. Определим для начала, какие поля нам необходимы для хранения информации о студенте. Требуемая информация представлена в таблице:

Фамилия	Имя	Возраст студента	Название университета	Год образования университета

Однако студентов, данные о которых будут храниться в базе данных, намного больше количества имеющихся университетов. И в результате может получиться, что из 10 000 зарегистрированных в базе студентов 5000 учатся в одном университете. А это значит, что в графе «Название университета» существует 5000 однотипных записей. То же самое и в графе «Год образования университета».

Если же мы хотим сэкономить на объеме базы данных, то нужно вынести поля, касающиеся университета, в отдельную таблицу. В результате мы получим две таблицы: одну с данными о фамилии, имени и возрасте студента («Студенты»), вторую — об университете, в котором он учится («Университеты»).

Теперь поля таблицы «Университеты» будут содержать намного меньше записей, чем когда они находились в исходной таблице. Возникает вопрос, как теперь мы узнаем, в каком именно из университетов учится тот или иной студент. Для этого в каждую из таблиц «Студенты» и «Университеты» нам необходимо добавить еще по одному полю, которые и будут определять связь между студентами и их учебными заведениями.

Фамилия	Имя	Возраст	ГО университета

Название университета	Год образования университета	Ш университета

Теперь, внося в базу информацию о студенте, не надо указывать полное название его учебного заведения, достаточно лишь знать ID нужного университета.

Например, в базе данных собрана информация о студентах, которые учатся в:

- Московском государственном технологическом университете;
- Московском государственном институте международных отношений;
- Московском государственном университете физкультуры и спорта.

Каждый из этих университетов имеет свой уникальный номер (ID). Соответственно:

- 1002;
- 1003;
- 1004.

Если студент учится в Московском государственном технологическом университете, то при внесении его в базу в поле «ID университета» мы запишем 1002.

Можно посчитать, сколько места на диске мы сэкономили, применив реляционную концепцию баз данных. Название «Московский государственный технологический университет» имеет размер 53 байта. Плюс поле «Год образования университета» имеет формат date/time и занимает 8 байт. 53 + 8 = 61. А число 1003 занимает всего 2 байта.

61 - 2 = 59. При наличии в базе 1000 студентов этого университета мы высвобождаем 59 х 1000 байт = 59 000 байт. С первого взгляда это небольшой объем. Но при осуществлении серьезных проектов эта цифра возрастает в сотни раз, что заметно сказывается на скорости обработки информации. Хотя если встает вопрос между выбором скорости и объема, следует отдавать предпочтение скорости, так как время доступа к данным на сегодняшний день намного дороже, чем цена жесткого диска.

Таблицы записи и поля

Теперь давайте рассмотрим реляционную модель данных на конкретном примере. Для этого возьмем базу данных Northwind, поставляемую вместе с SQL server 7.0 или SQL server 2000. Northwind — это база данных, описывающая вымышленную компанию, покупающую и продающую продовольственные товары. Данные для Northwind разделены на 13 таблиц, таких как Customers, Employees, Orders, Order Details, Products и т. д.

Каждая таблица в реляционной базе данных организована в строки, где каждая строка представляет собой отдельную запись. Строки организованы в столбцы. Все строки в таблице имеют одинаковую структуру столбцов (полей). Например, таблица заказов Orders имеет следующие поля: OrderID, CustomerlD, EmployeeID, OrderDate и т. д. Для любого заказа вы должны знать имя заказчика, его адрес, контактное имя и т. д. Вы можете сохранять информацию для каждой записи, но это будет неэффективно. Вместо этого создатели базы придумали вторую таблицу с именем Customers, в которой каждая строка представляет собой единственного заказчика. Каждый заказчик имеет уникальный идентификатор (поле CustomerlD, которое отмечено как первичный ключ для этой таблицы). Первичный ключ — это столбец или комбинация столбцов, которые уникально идентифицируют запись в таблице.

Таблица Orders использует поле CustomerlD как вторичный ключ (foreign key). Вторичный ключ — это поле или комбинация полей, которые являются первичным ключом для другой таблицы. Таблица Orders использует CustomerlD (первичный ключ, используемый в таблице Customers), чтобы

выделить, какой именно клиент оформил заказ. Чтобы определить адрес заказа, вы можете использовать CustomerID, который поможет вам найти заказчика в таблице Customers.

Использование вторичных ключей особенно эффективно в таблицах, связанных отношениями one-to-many или many-to-one. Разделяя информацию на таблицы, которые связаны вторичными ключами, вы избавляете себя от необходимости повторять информацию в своих записях. Один заказчик, например, может иметь несколько заказов. Но тогда будет неэффективно размещать информацию о заказчике (имя, номер телефона, сумму кредита и т. д.) в каждой записи. Процесс удаления избыточной информации из полей таблицы и размещения их в отдельных таблицах называется *нормализацией.*

Нормализация

Нормализация делает использование вашей базы данных не только более эффективным, но также уменьшает вероятность нарушения целостности данных. Если бы вы хранили имя заказчика и в таблице Customers, и в таблице Orders, то рисковали бы тем, что изменение данных в одной таблице повлечет за собой сбой в другой таблице. То есть если вы измените имя заказчика в таблице Customers, это не вызовет автоматического изменения имени в каждой строке таблицы Orders. Следовательно, появится несоответствие между таблицами, и вам придется проделать много работы, чтобы внести все необходимые изменения в таблицу Orders. Если же вы храните только CustomerID в таблице Orders, то можете изменять адрес в таблице Customers без дополнительных действий, поскольку такое изменение никак не влияет на поля таблицы Orders.

SQL сервер и другие современные СУБД позволяют избежать ошибок в базах данных, предписывая ограничения на запросы, которые делает программист. Например, таблица Customers в базе данных Northwind содержит поле CustomerID, которое является первичным ключом. Это накладывает ограничение на базу данных, которое гарантирует, что каждый CustomerID уникален. Если у вас имеется заказчик с именем «Алексей Рубинов», у которого CustomerID=32, то вы не сможете добавить нового заказчика «Василий Лазерко» с CustomerID=32. База данных отклонит ваш запрос на добавление новой записи, потому что первичный ключ должен быть уникальным.

ЯЗЫК SQL И ПРЕДЛОЖЕНИЕ SELECT

Все запросы на получение практически любого количества данных из одной или нескольких таблиц выполняются с помощью единственного предложения SELECT. В общем случае результатом реализации предложения SELECT является другая таблица. К этой новой (рабочей) таблице может быть снова применена операция SELECT и т.д., то есть такие опе-

рации могут быть вложены друг в друга. Представляет исторический интерес тот факт, что именно возможность включения одного предложения SELECT внутрь другого послужила мотивировкой использования прилагательного «структуризированный» в названии языка SQL.

Предложение SELECT может использоваться как:

- самостоятельная команда на получение и вывод строк таблицы, сформированной из столбцов и строк одной или нескольких таблиц (представлений);
- элемент WHERE- или HAVING-условия (сокращенный вариант предложения, называемый «вложенный запрос»);
- фраза выбора в командах CREAT VIEW, DECLARE CURSOR или INSERT;
- средство присвоения глобальным переменным значений из строк сформированной таблицы (INTO-фраза).

Основные обозначения, используемые в предложении SELECT

В данной главе будут рассмотрены только две первые функции предложения SELECT. Здесь в синтаксических конструкциях используются следующие обозначения:

- звездочка (*) для обозначения «все» — употребляется в обычном для программирования смысле, т. е. «все случаи, удовлетворяющие определению»;
- квадратные скобки ([]) означают, что конструкции, заключенные в эти скобки, являются необязательными (т. е. могут быть опущены);
- фигурные скобки ({}) означают, что конструкции, заключенные в эти скобки, должны рассматриваться как целые синтаксические единицы, т. е. они позволяют уточнить порядок разбора синтаксических конструкций, заменяя обычные скобки, используемые в синтаксисе SQL;
- многоточие (...) указывает на то, что непосредственно предшествующая ему синтаксическая единица факультативно может повторяться один или более раз;
- прямая черта (|) означает наличие выбора из двух или более возможностей. Например, обозначение ASC|DESC указывает: можно выбрать один из терминов ASC или DESC; когда же один из элементов выбора заключен в квадратные скобки, то это означает, что он выбирается по умолчанию (так, [ASC]|DESC означает, что отсутствие всей этой конструкции будет восприниматься как выбор ASC);
- точка с запятой (;) — завершающий элемент предложений SQL;
- запятая (,) используется для разделения элементов списков;
- пробелы ( ) могут вводиться для повышения наглядности между любыми синтаксическими конструкциями предложений SQL;
- прописные латинские буквы и символы используются для написания конструкций языка SQL;
- строчные буквы используются для написания конструкций, которые должны заменяться конкретными значениями, выбранными пользова-

телем, причем для определенности отдельные слова этих конструкций связываются между собой символом подчеркивания (_);
- термины «таблица», «столбец», ... заменяют (с целью сокращения текста синтаксических конструкций) термины «имя_таблицы», «имя_столбца», ..., соответственно;
- термин таблица используется для обобщения таких видов таблиц, как базовая_таблица, представление или псевдоним; здесь псевдоним служит для временного (на момент выполнения запроса) переименования и (или) создания рабочей копии базовой_таблицы (представления).

Формат предложения SELECT

SELECT (выбрать) данные из указанных столбцов и (если необходимо) выполнить перед выводом их преобразование в соответствии с указанными выражениями и (или) функциями.

FROM (из) перечисленных таблиц, в которых расположены эти столбцы.

WHERE (где) строки из указанных таблиц должны удовлетворять указанному перечню условий отбора строк.

GROUP BY (группируя по) указанному перечню столбцов с тем, чтобы получить для каждой группы единственное агрегированное значение, используя во фразе SELECT SQL-функции SUM (сумма), COUNT (количество), MIN (минимальное значение), MAX (максимальное значение) или AVG (среднее значение).

HAVING (имея) в результате лишь те группы, которые удовлетворяют указанному перечню условий отбора групп.

Фраза WHERE включает набор условий для отбора строк:

```
WHERE [NOT] WHERE_условие [ [AND|OR] [NOT] WHERE_условие] . . .
```

где WHERE_условие имеет следующую конструкцию:

значение1 { = I o I < I <= I > 1 >= } значение2.

Кроме традиционных операторов сравнения (= I <> 1 < I <= I > I >=), в WHERE фразе используются условия:
- BETWEEN (между);
- LIKE (похоже на);
- IN (принадлежит);
- IS NULL (не определено);
- EXISTS (существует),

которые могут предваряться оператором NOT (не). Критерий отбора строк формируется из одного или нескольких условий, соединенных логическими операторами:

AND — когда должны удовлетворяться оба разделяемых с помощью AND условия;

OR —когда должно удовлетворяться одно из разделяемых с помощью OR условий;

AND NOT —когда должно удовлетворяться только первое условие;

OR NOT —когда или должно удовлетворяться первое условие, или не должно удовлетворяться второе, причем существует приоритет AND над

OR (сначала выполняются *все* операции AND и только после этого операции OR). Для получения желаемого результата WHERE условия должны быть введены в правильном порядке, который можно организовать введением скобок.

При обработке условия числа сравниваются алгебраически. Отрицательные числа считаются меньшими, чем положительные, независимо от их абсолютной величины. Строки символов сравниваются в соответствии с их представлением в коде, используемом в конкретной СУБД, например в коде ASCII. Если сравниваются две строки символов, имеющих разные длины, более короткая строка дополняется справа пробелами для того, чтобы они имели одинаковую длину перед осуществлением сравнения.

Наконец, синтаксис фразы GROUP BY имеет вид

```
GROUP BY [таблица.] столбец [,[таблица.] столбец]... [HAVING фраза]
```

GROUP BY инициирует перекомпоновку формируемой таблицы по группам, каждая из которых имеет одинаковое значение в столбцах, включенных в перечень GROUP BY. Далее к этим группам применяются агрегирующие функции, указанные во фразе SELECT, что приводит к замене всех значений группы на единственное значение (сумма, количество и т. п.).

С помощью следующей инструкции можно исключить из результата те группы, которые удовлетворяют заданным условиям.

```
[NOT] HAVING_условие [[AND|OR][NOT] HAVING_условие]
```

МОДЕЛЬ ОБЪЕКТОВ ADO.NET

DataSet

Модель объектов ADO.NET очень обширна, но в ее основе лежит довольно простой набор классов. Наиболее важным из них считается DataSet. DataSet представляет собой отображение используемой базы данных, перенесенное на машину пользователя. При этом нет необходимости постоянно подключаться к серверу базы данных для модификации данных.

Лишь иногда вы соединяете DataSet с его родительской базой данных и модифицируете ее внесенными вами изменениями. В то же время вы модифицируете DataSet теми изменениями в базе данных, которые сделали другие процессы.

DataSet состоит из объектов типа DataTable и объектов DataRelation. К ним можно обращаться как к свойствам объекта DataSet. Свойство Tables возвращает объект типа DataTableCollection, который содержит все объекты DataTable используемой базы.

Таблицы и поля (объекты DataTable и DataColumn)

Объект типа DataTable представляет собой таблицу базы данных. Такой объект может быть создан программно или путем запроса к базе данных. Объект DataTable состоит из строк и столбцов. Строки представ-

ляют собой отдельные записи таблицы, столбцы — соответствующие поля. Для получения совокупности столбцов объект DataSet имеет свойство Columns, возвращающее DataColumnCollection, которое в свою очередь состоит из объектов типа DataColumn. Каждый объект DataColumn представляет собой отдельный столбец таблицы, из которого можно получить любую запись.

Связи между таблицами (объект DataRelation)

Кроме набора таблиц DataSet имеет свойство Relations, которое возвращает объект типа DataRelationCollection, состоящий из объектов DataRelation. Каждый DataRelation объект хранит данные о связях между двумя таблицами посредством объектов DataColumn. Например, в базе данных Northwind таблица Customers имеет связь с таблицей Orders посредством столбца CustomerID. Такое отношение называется на языке баз данных *один ко многим* (one-to-many). Для любого заказа может быть только один заказчик, но один заказчик может иметь сколько угодно заказов.

Строки (объект DataRow)

Свойство Rows объекта DataTable возвращает совокупность всех строк таблицы — DataRowCollection. Это свойство следует применять для того, чтобы пользоваться результатами запросов к базе данных. Программисты, имеющие опыт работы с ADO, будут удивлены отсутствием RecordSet с его функциями moveNext и movePrevious. В ADO.NET нет необходимости в итерационном обходе DataSet для получения данных. Вы можете обращаться к записям таблицы как к элементам простого массива. Это значительно упрощает процесс доступа к элементам базы. Мы рассмотрим это более подробно далее на примере.

DataAdapter

DataSet — это образ реляционной базы данных. ADO.NET использует объект типа DataAdapter как мост между DataSet и источником данных, который является основной базой данных. DataAdapter содержит метод Fill() для обновления данных из базы и заполнения DataSet.

DBCommand и DBConnection

Объект DBConnection представляет собой средство для соединения DataSet с источником данных. Соединение может быть доступно при помощи различных командных объектов. Например, объект DBCommand позволяет послать команду (обычно это SQL запрос или сохраненная процедура) к базе данных. Часто командные объекты создаются неявно, во время формирования объекта DataSet. Но ADO.NET позволяет вам явно обращаться к таким объектам, это будет рассмотрено в примере.

РАБОТА С ADO.NET

Кажется, уже пришло время закрепить теоретические знания на практике. Давайте напишем пример программы, использующей ADO.NET, и посмотрим, как это все работает. В этом примере мы создадим простое Windows Forms приложение с единственным списком в окне и заполним этот список информацией из таблицы Customers базы данных Northwind.

Я предлагаю вам два варианта создания приложения для работы с базами банных. Первый из них более подходит для тех, кто не имеет большого опыта работы на C#. Этот способ подразумевает использование визуальной среды Visual Studio .NET для создания объектов. Второй способ — это создание объектов непосредственно в коде программы, что, согласитесь, требует определенной подготовки.

Использование визуальной среды для работы с ADO.NET

Создайте новое приложение Windows Forms с именем ADOWinForms. Разместите на создавшейся форме компонент ListBox и измените его свойство name на listCustomers. Теперь в окне *ToolBox* перейдите на закладку *Data* (см. рис. 33.1).

Поместите на форму компонент SqlDataAdapter. Перед вами появится окно дизайнера для настройки компонента. Нажмите в появившемся окне кнопку *Next*. Следующее окно предлагает выбрать источник для соединения с базой данных. В нашем случае еще не существует ни одного источника, поэтому нажмите кнопку *New Connection...* Теперь нам предстоит создать новый источник соединения С базой даННЫХ.

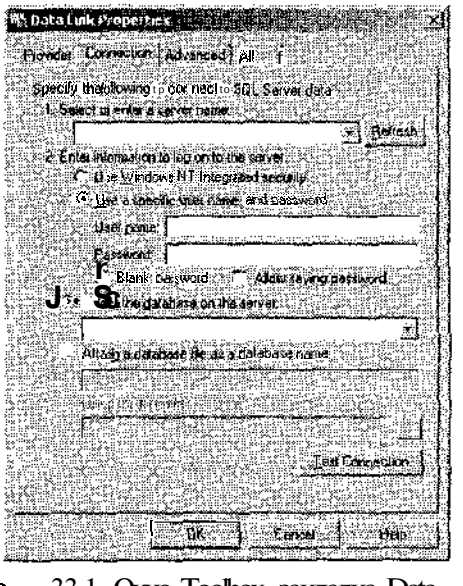


Рис. 33.1. Окно Toolbox, закладка Data

Соединение с сервером

Первым делом необходимо выбрать имя сервера базы данных. Для этого в окне, изображенном на рис. 33.1, имеется поле *Select or enter server name*. Здесь вам необходимо выбрать один из доступных вашей машине серверов. Если у вас на машине установлен и запущен SQL server, то выберите в списке имя своей машины, если нет, то вам предстоит установить его.

Этот абзац предназначен для тех, у кого на машине не установлен SQL Server. Проще всего в таком случае выбрать пункт меню *Samples and Quickstart Tutorials* из *Microsoft .NET Framework SDK* группы программ и

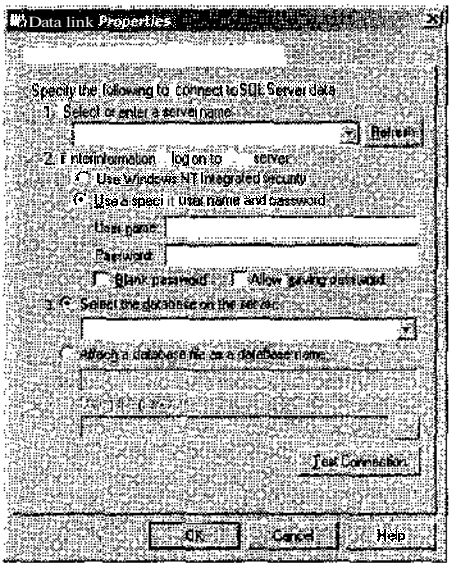


Рис. 33.2. Настройка нового соединения с базой данных

выполнить требования по установке .NET Framework Samples Database, которая включает инсталляцию SQL сервера. После установки базы данных с примерами, установите QuickStarts (установится база данных northwind). Теперь в окне, изображенном на рис. 33.2 в поле *Select or enter server name* выберите строку localhost\\NetSDK, где localhost — это имя машины, на которой установлен сервер базы данных. Либо введите строку («(local)WNetSDK»), в этом случае программа сама определит имя вашей машины (это не совсем корректное определение, но оно хорошо поясняет строку данных).

В поле *Enter Information to logon to the server* выберите пункт *Use Windows NT Integrated Security*. Это даст вам возможность подключаться к базе, используя ваше имя для работы в Windows. Вы можете использовать любое известное вам имя и пароль для подключения к базе. В таком случае, необходимо выбрать пункт *Use a specific user name and password* и прописать соответствующие значения в полях *User Name* и *Password*.

Если вы правильно настроили права доступа к базе данных, то поле *Select the Database on the server* даст вам возможность выбрать одну из существующих на сервере базу данных. Нас интересует база Northwind. Выберите ее и нажмите кнопку *Test Connection*. В случае успешного соединения система выдаст сообщение *Test Connection succeeded,* что означает успешное соединение с базой. Нажмите *OK* для перехода к следующему окну настройки DataAdapter.

Доступ к данным

Перед вами появится окно выбора способа доступа к элементам базы данных (см. рис. 33.3).

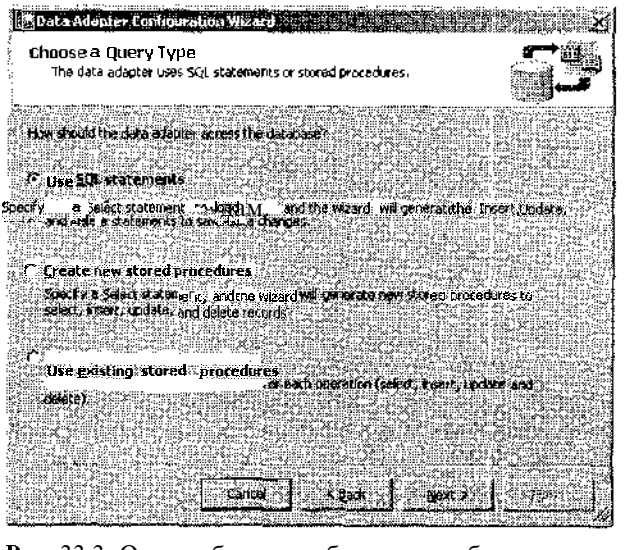


Рис. 33.3. Окно выбора способа доступа к базе данных

Этот диалог дает вам возможность выбрать один из трех способов доступа к базе:

- используя SQL инструкции;
- создав новую сохраненную процедуру (storage procedure);
- используя существующую сохраненную процедуру.

Мы воспользуемся наиболее простым способом доступа к базе, прибегнув к SQL запросу. Для этого выберите в диалоге пункт *Use SQL statements*.

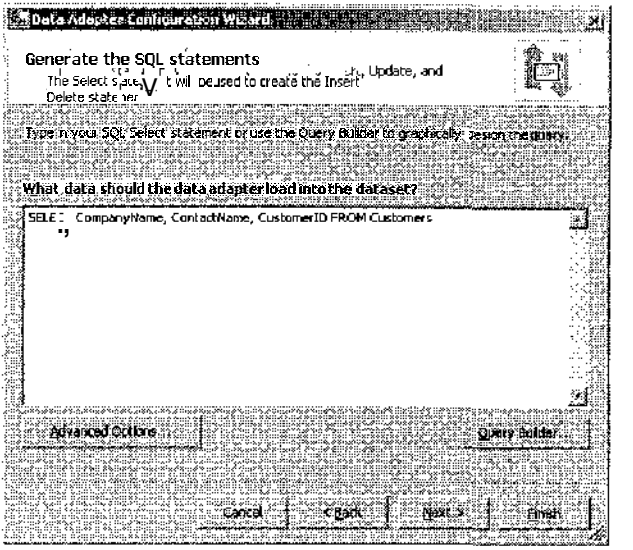


Рис. 33.4. Окно создания SQL запроса

Создание SQL запроса

В поле для редактирования окна *Generate SQL statements* введите SQL инструкцию: «Select CompanyName, ContactName from Customers» (см. рис. 33.4).

В этом же окне нажмите кнопку *Advanced Options* и уберите все флажки в открывшемся диалоге. Это избавит вас от массы ненужного в данной программе кода. Подтвердите все сделанные изменения и перейдите в окно *View Wizard Results*. Оно должно иметь вид, аналогичный представленному на рис. 33.5.

Это означает, что мастер создаст для вас SQL инструкцию и образ таблицы базы данных (речь о нем пойдет далее в этой главе).

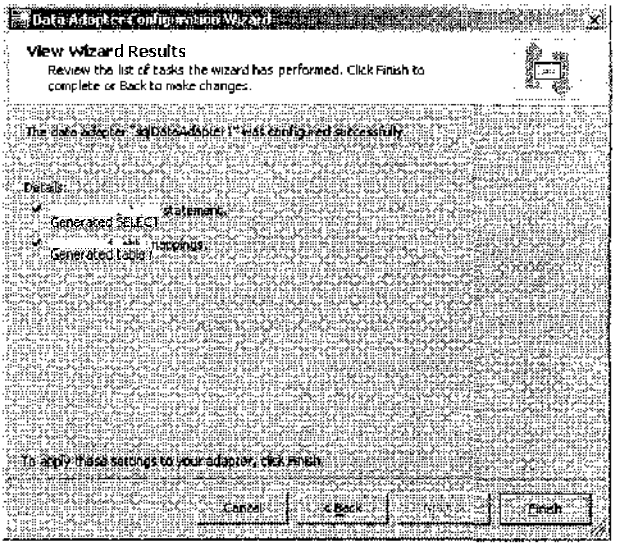


Рис. 33.5. Окно результатов настройки компонента DataAdapter

Что же сделал мастер работы с базами данных?

Как вы можете заметить, мастер создал для вас в панели компонент два элемента: sqlDataAdapter1 и sqlConnection1. sqlDataAdapter1 является основным компонентом типа SqlDataAdapter, который мы создавали при по-

мощи мастера. sqlConnectionl — это объект типа SqlConnection, описанный мною ранее в этой главе в пункте «DBCommand и DBConnection». Он создался мастером как необходимый элемент для подключения к базе. Давайте изменим имена наших элементов на более читаемые: dataAdapter и connection соответственно.

Анализ кода программы

Теперь обратимся к коду нашей программы:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace ADOWinForms
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form
  {
    private System.Windows.Forms.ListBox listCustomers;
    private System.Data.SqlClient.SqlCommand sqlSelectCommand1;
    private System.Data.SqlClient.SqlConnection connection;
    private System.Data.SqlClient.SqlDataAdapter dataAdapter;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
```

```
    protected override void Dispose( bool disposing )
    {
      if ( disposing )
      {
        if (components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose ( disposing );
    }

#region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    ///  the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
      this.listCustomers = new System.Windows.Forms.ListBox();
      this.dataAdapter = new System.Data.SqlClient.SqlDataAdapter();
      this.sqlSelectCommand1 = new System.Data.SqlClient.SqlCommand();
      this.connection = new System.Data.SqlClient.SqlConnection();
      this.SuspendLayout();
      //
      // listCustomers
      //
      this.listCustomers.Location = new System.Drawing.Point(8, 8);
      this.listCustomers.Name = "listCustomers";
      this.listCustomers.Size = new System.Drawing.Size(280, 264);
      this.listCustomers.TabIndex = 0;
      //
      // dataAdapter
      //
      this.dataAdapter.SelectCommand = this.sqlSelectCommand1;
      this.dataAdapter.TableMappings.AddRange(
        new System.Data.Common.DataTableMapping[]
      {
        new System.Data.Common.DataTableMapping("Table", "Customers",
          new System.Data.Common.DataColumnMapping[]
        {
          new System.Data.Common.DataColumnMapping("CompanyName",
          "CompanyName"),
            new System.Data.Common.DataColumnMapping("ContactName",
            "ContactName")
        }
```

```
        }
      }
      );
      //
      // sqlSelectCommand1
      //
      this.sqlSelectCommand1.CommandText =
        "SELECT CompanyName, ContactName, CustomerID FROM Customers";
      this.sqlSelectCommand1.Connection = this.connection;
      //
      // connection
      //
      this.connection.ConnectionString =
        "data source=(local)\\NetSDK;initial catalog=Northwind;"+
        "integrated security=SSPI;persist security info=False;"+
        "user id=sa;workstation id=(local);packet size=4096";
      //
      // Form1
      //
      this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
      this.ClientSize = new System.Drawing.Size(292, 273);
      this.Controls.AddRange(new System.Windows.Forms.Control[]
      {
        this.listCustomers
      });
      this.Name = "Form1";
      this.Text = "Form1";
      this.ResumeLayout(false);

    }
#endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
  }
}
```

Рассмотрим более подробно некоторые участки кода программы.

```
private System.Windows.Forms.ListBox listCustomers;
```

Создание визуального компонента ListBox с именем listCustomers.

```
private System.Data.SqlClient.SqlCommand sqlSelectCommand1;
```

Объявление объекта SqlCommand с именем sqlSelectCommand1. Этот объект будет использоваться программой как команда для получения доступа к базе данных. sqISelectCommandl был создан мастером при настройке компонента DataAdapter.

```
private System.Data.SqlClient.SqlConnection connection;
```

Объявление объекта типа SqlConnection с именем connection. SqlConnection объект используется для соединения с базой данных. Этот объект предназначен для хранения данных о сервере базы данных, пользователе, пароле пользователя, названии базы ...

```
private System.Data.SqlClient.SqlDataAdapter dataAdapter;
```

Объявление объекта типа SqlDataAdapter с именем dataAdapter.

```
his.listCustomers = new System.Windows.Forms.ListBox();
this.dataAdapter = new System.Data.SqlClient.SqlDataAdapter();
this.sqlSelectCommand1 = new System.Data.SqlClient.SqlCommand();
this.connection = new System.Data.SqlClient.SqlConnection();
```

Создание объектов listCustomers, dataAdapter, sqISelectCommandl и connection. Заметьте, что именно здесь происходит создание объектов, т. е. вызов оператора new, до этого они были только объявлены.

```
this.dataAdapter.SelectCommand = this.sqlSelectCommand1;
```

DataAdapter имеет свойство SelectCommand. Это свойство должно быть установлено до того, как объект DataAdapter будет использоваться.

```
//настройка образа таблицы используемой адаптером
this.dataAdapter.TableMappings.AddRange(
  //создание массива образов таблицы
  new System.Data.Common.DataTableMapping[]
  {
    //инициализация образа одним объектом
    new System.Data.Common.DataTableMapping(
      "Table", //имя таблицы используемой в программе
      "Customers", //имя таблицы в базе данных
      //инициализация образа полей таблицы
      new System.Data.Common.DataColumnMapping[]
      {
        //создание образа поля CompanyName
        new System.Data.Common.DataColumnMapping(
          "CompanyName", //имя поля в программе
          "CompanyName"}, //имя поля в базе данных
        //создание образа поля ContactName
        new System.Data.Common.DataColumnMapping(
          "ContactName", //имя поля в программе
          "ContactName"} //имя поля в базе данных
      }
  )
}
);
```

Я позволил себе немного отформатировать следующий участок кода, и сделал это для того, чтобы дать в комментариях объяснение его предназ-

начения. Хочу заметить, что программа будет работать и без этого кода. Но данный фрагмент позволяет программисту работать с базой данных в интуитивно понятной форме. Для этого создается образ (его еще называют отображением (Mapping) таблиц базы данных в программе. Образ позволит вам обращаться к таблицам и полям базы данных по именам, что гораздо удобнее, нежели использование индексов. Мы еще рассмотрим использования отображения в следующих примерах.

```
this.sqlSelectCommand1.CommandText="SELECT CompanyName, ContactName, CustomerID
FROM Customers";
this.sqlSelectCommand1.Connection= this.connection;
```

Объекты типа SqlCommand имеют свойство CommandText. Это свойство хранит строку SQL для доступа к данным. Вы можете видеть, что этому свойству присваивается строка, заданная нами в окне, изображенном на рис. 33.5. Другим важным свойством компонента SqlCommand является свойство Connection. Оно связывает SqlCommand с объектом SqlConnection.

```
this.connection.ConnectionString = "data source=(local)\\NetSDK;initial
catalog=Northwind;integrated security=SSPI;persi" +
"st security info=False;user id=sa;workstation id=(local);packet size=4096";
```

Инициализация свойства ConnectionString объекта SqlConnection. Свойство ConnectionString представляет собой простую строку. Эта строка имеет строго определенный формат. В ней вам необходимо задать параметры соединения с базой данных. Все параметры передаются одной строкой и разделяются знаком «;». Список используемых параметров:

- data source — сетевое имя сервера базы данных;
- initial catalog — имя базы данных на сервере;
- integrated security — способ доступа к базе;
- user id — имя пользователя для входа в базу (sa по умолчанию);
- workstation id — имя или IP адрес компьютера, с которого подсоединяются к базе.

Расширение функциональности программы

Весь этот код был создан дизайнером форм и мастером работы с базами данных. Объем кода представляется довольно значительным. Однако вам не приходится об этом задумываться, ведь за вас все сделали разработчики Visual Studio .NET. Откомпилируйте и запустите программу. Вы получите окно с пустым списком данных. «Но где же данные из базы?» — спросите вы. Это нам придется написать самим в коде программы: нам предстоит перенести данные, прочитанные из базы данных, в список listCustomers. Для этого создайте обработчик события Load формы. Щелкните два раза левой кнопкой мыши по имени этого события в окне *Properties*. Добавьте в функцию Form1_Load код, представленный ниже:

```
private void Form1_Load (object sender, System.EventArgs e)
    {
     //создаем новый объект DataSet
     DataSet ds = new DataSet();
```

```
    //вызываем функцию Fill для заполнения объекта
    //DataSet содержимым таблицы Customers
    dataAdapter.Fill(ds, "Customers");

    //получаем доступ к таблице Customers
    DataTable dt = ds.Tables["Customers"];

    //переносим данные в список
    //"Контактное имя" работает на "организация"
    //например Иван Петров работает на "СП СибирьНефть"
    foreach (DataRow dataRow in dt.Rows)
    {
      listCustomers.Items.Add(dataRow["ContactName"] +
        " работает на \"" + dataRow["CompanyName"] + "\"");
    }
  }
```

Код, который создала среда Visual Studio .NET, не содержит методов для чтения данных из базы. Для подключения и прочтения данных используется метод Fill() класса SqlDataAdapter.

```
dataAdapter.Fill(ds, "Customers");
```

При вызове этого метода происходит заполнение объекта DataSet содержимым базы данных, на которую настроен DataAdapter. В нашем случае одним из параметров метода Fill() является имя таблицы, которое указывает, какой именно образ следует брать для отображения таблицы. При вызове этого метода DataAdapter уже должен быть проинициализирован всеми необходимыми параметрами. В обязательном порядке должно быть установлено свойство SelectCommand. При возникновении неопределенностей при обращении к базе программа генерирует исключения, которые вы должны обрабатывать в реальных приложениях.

Для удобства работы программиста с таблицами базы данных предназначен класс DataTable. Он содержит методы и данные для обработки любых табличных данных (не только из базы данных). Объект DataSet хранит таблицы как массив данных. Программист может обращаться к таблицам по их индексам. Например, ds.Tables[0] — обращение к нулевой таблице. Однако, согласитесь, это не очень удобно. Какая таблица имеет индекс 0? Вам постоянно нужно помнить об этом. Но если таблиц десятки, а то и сотни, что делать тогда? Вот для этого разработчики C# и придумали отображение таблицы, которое мы создали для таблицы Customers. Теперь мы можем использовать эту таблицу по ее именованному индексу «Customers». Вот так мы определили объект dt таблицей «Customers» объекта ds.

```
DataTable dt = ds.Tables["Customers"];
```

Таблица состоит из строк и столбцов. Строки представляют собой записи таблицы, столбцы — поля таблицы. Для доступа к строкам используется свойство Rows. Это свойство возвращает объект типа DataRowCollection. За счет реализации этим классом интерфейса ICollection мы можем обращаться к строкам с использованием оператора foreach.

```
foreach (DataRow dataRow in dt.Rows)
{
  listCustomers.Items.Add( dataRow["ContactName"] +
    " работает на \"" + dataRow["CompanyName"] + "\"");
}
```

Еще раз хочется отметить преимущества использования отображений. При обращении к полям таблицы мы указываем не индекс поля в общем массиве полей, а название поля, например dataRow["ContactName"], что более понятно при анализе кода.

Результаты работы программы

Теперь нашу программу можно считать законченной. Откомпилируйте и запустите ее. На экране должно появиться окно, аналогичное изображенному на рис. 33.6.

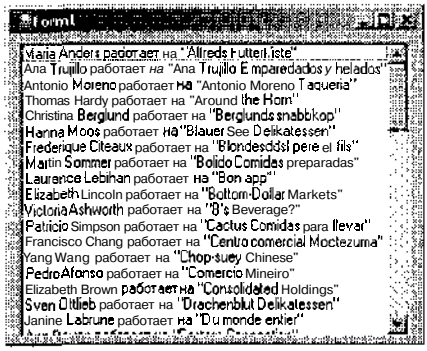


Рис. 33.6 Окно информации о контактных лицах

Программирование компонент баз данных

Мы рассмотрели процесс создания программы для работы с базами данных с использованием различных мастеров Visual Studio .NET. Давайте рассмотрим, как работать с базами данных без визуальной среды. Создадим такое же приложение, как и в предыдущем примере.

Для начала необходимо включить в программу объект типа DataAdapter.

```
SqlDataAdapter dataAdapter =
    new SqlDataAdapter( commandString, connectionString);
```

Здесь используется иной конструктор, чем в предыдущем примере, где мы не инициализировали DataAdapter. Первым параметром передается строка, содержащая SQL инструкцию для доступа к данным. Она должна быть объявлена следующим образом:

```
string commandString =
  "Select ContactName, CompanyName from Customers";
```

Вторая строка описывает способ доступа к базе. Поскольку мы используем ту же базу, что и в предыдущем примере, я не буду приводить подробного описания формата строки. Кроме того, я не стану форматировать connectionString необязательными для нашего примера параметрами.

```
string connectionString =
  "server=(local)\\NetSDK; uid=sa;pwd=;database=Northwind";
```

Теперь у вас есть DataAdapter, и вы можете употребить его для заполнения DataSet, используя уже имеющуюся SQL инструкцию.

```
DataSet dataSet = new DataSet( );
dataAdapter.Fill(dataSet);
```

Заметьте, что мы не применяем отображения. Посмотрим, чем нам придется пожертвовать из-за этого.

Имея заполненный DataSet, вы можете получить доступ к прочитанным данным. Читаем таблицу:

```
DataTable dataTable = DataSet.Tables[0];
```

Для чтения таблицы необходимо использовать ее индекс — нулевой, поскольку только одна таблица была прочитана. Вот это и есть первый недостаток отказа от отображений.

Затем необходимо извлечь данные из таблицы и поместить их в список.

```
foreach (DataRow dataRow in dataTable.Rows)
{
  listCustomers.Items.Add(
    dataRow[0] + "работает на \"" + dataRow[1] -r "\"" );
}
```

Здесь для обращения к полям таблицы используются не имена, а номера полей в массиве. Поля располагаются в порядке их выборки из базы, то есть в очередности записи в SQL запросе. В нашем случае поле «ContactName» имеет индекс 0, а поле «CompanyName» индекс 1.

Вот полный листинг программы.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace ADOWinForms
{
  public class Form1: System.Windows.Forms.Form
  {
    private System.ComponentModel.Container components;
    private System.Windows.Forms.ListBox listCustomers;

    public Form1( )
    {
      InitializeComponent( );

      // connect to my local server, northwind db
      string connectionString = "server= (local)\\NetSDK; " +
        "Trustedconnection=yes; database=northwind";

      // get records from the customers table
      string commandString =
        "Select ContactName, CompanyName from Customers";

      // create the data set command object
```

```
    // and the DataSet
    SqlDataAdapter dataAdapter =
      new SqlDataAdapter(
      commandString, connectionString);

    DataSet dataSet = new DataSet ( ) ;

    // fill the data set object.
    dataAdapter.Fill(dataSet,"Customers");

    // Get the one table from the DataSet
    DataTable dataTable = dataSet.Tables[0];

    // for each row in the table, display the info
    foreach (DataRow dataRow in dataTable.Rows)
    {
      listCustomers.Items.Add(
        dataRow[0] +
        " работает на \"" + dataRow[1] + "\"" );
    }
  }

  private void InitializeComponent( )
  {
    this.listCustomers = new System.Windows.Forms.ListBox();
    this.SuspendLayout();
    //
    // listCustomers
    //
    this.listCustomers.Location = new System.Drawing.Point(8, 8);
    this.listCustomers.Name = "listCustomers";
    this.listCustomers.Size = new System.Drawing.Size(368, 225);
    this.listCustomers.TabIndex = 0;
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(384, 245);
    this.Controls.AddRange(new System.Windows.Forms.Control[] {
        this.listCustomers});

    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
  }
```

```
    public static void Main(string[] args)
    {
      Application.Run(new Form1( ));
    }
  }
}
```

Откомпилировав и запустив программу, вы получите тот же результат, что и в предыдущем примере — окно, изображенное на рис. 33.6.

Как можно было заметить, программа, созданная без использования мастеров Visual Studio .NET, короче. Однако ее текст менее читаемый. Вы вольны выбирать — работать мастером баз данных или писать код самому. Я советую вам первое время использовать возможности визуальной среды для разработки приложений. Когда вы почувствуете себя специалистом в программировании на C#, то сможете с легкостью от них отказаться.

ИСПОЛЬЗОВАНИЕ OLE DB ДЛЯ ДОСТУПА К ДАННЫМ

Мы рассмотрели лишь один из способов работы с базами данных, предлагаемый разработчиками C#,— это использование «SQL server provider». Данный способ предполагает обязательное применение SQL server для работы с базами данных. Я не говорю, что Microsoft навязывает использование SQL server в качестве основного сервера баз данных при разработке приложений на C#, но его разработчики не поленились выделить SQL server в отдельную категорию компонент.

Второй возможностью доступа к данным является использование OLE DB Provider. Такой способ позволяет осуществлять доступ к любому OLE DB провайдеру, включая Oracle, Sybase и, конечно же, Access. Мы рассмотрим пример работы OLE DB провайдера с использованием базы данных Access.

Для создания рабочего примера приложения вам необходимо экспортировать базу данных Northwind в файл mdb формата. Для этого воспользуйтесь руководством по SQL server. Я экспортировал базу в файл под названием «C:\nothwind.mdb».

Давайте создадим приложение, наделенное той же функциональностью, что и в предыдущем примере. Программа будет читать данные о компаниях и контактных лицах компаний и выводить информацию в список.

Возможности Visual Studio .NET при использовании OLE DB

При создании приложения для работы с базами данных с помощью OLE DB используется тот же мастер, что и при работе с SQL server. Однако имеются свои нюансы.

Создайте новое приложение Windows Forms с именем OLEDBWinForms. Разместите на создавшейся форме компонент ListBox и измените его свойство name на listCustomers. Теперь в окне *ToolBox* перейдите на закладку *Data* (см. рис. 33.1).

Поместите на форму компоненту OLEDBDataAdapter. Перед вами появится окно дизайнера для настройки компоненты. Нажмите в появившемся окне кнопку *Next*. Следующее окно предлагает выбрать источник для соединения с базой данных. Это окно открывается по умолчанию, поскольку предполагает соединение с использованием SQL сервера. Перед нами стоит другая задача: подключиться к базе с помощью OLE DB провайдера для баз ACCESS. Поэтому перейдите в окне, изображенном на рис. 33.2, на закладку *Provider* (рис. 33.7).

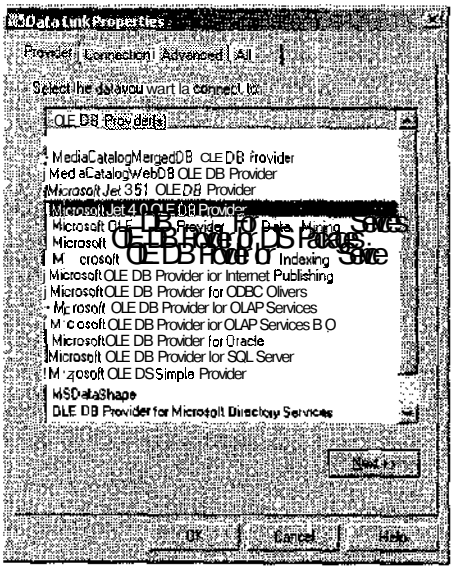


Рис. 33.7. Окно выбора провайдера

Выбор провайдера

По своей сути OLE DB является интерфейсом, посредством которого ваша программа может использовать СУБД сторонних разработчиков.

Нам необходим Microsoft Jet 4.0 OLE DB Provider. Выберите его, как показано на рис. 25.7. Он позволяет соединяться с базами формата Microsoft Access (mdb). Нажмите кнопку *Next* или перейдите на закладку *Connection* (рис. 33.8).

В поле *Select* or enter *database name* введите путь к файлу вашей базы данных. В моем случае это "C:\northwind.mdb". Если вы не изменяли права доступа к базе данных, то оставьте все настройки окна по умолчанию.

Если вы правильно настроили права доступа к базе данных, то при нажатии кнопки *Test Connection* система выдаст сообщение *Test Connection succeeded,* что означает успешное соединение с базой. Нажмите *OK* для перехода к следующему окну настройки DataAdapter.

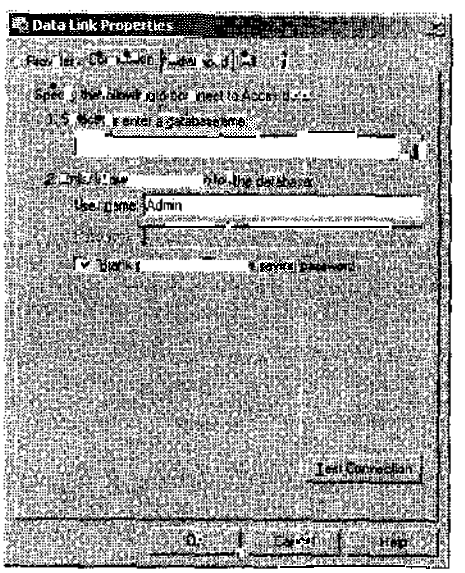


Рис 33.8. Окно подключения к базе Access

Доступ к данным

Перед вами появится окно выбора способа доступа к элементам базы данных (см. рис. 33.3). Но на этот раз у вас не будет выбора между SQL запросом и хранимой процедурой, поскольку Access не поддерживает хранимых процедур. Просто нажмите *Next.*

Создание SQL запроса

В поле для редактирования окна *Generate SQL statements* введите следующую SQL инструкцию: «Select CompanyName, ContactName from Customers» (см. рис. 33.4).

В этом же окне нажмите кнопку *Advanced Options* и уберите все флажки в открывшемся диалоге. Это избавит вас от массы ненужного в данной программе кода. Подтвердите сделанные изменения.

Результаты работы с мастером настройки базы данных

В окне *View Wizard Results* вы можете видеть тот же результат, что и при разработке примера прошлого приложения. Нажмите *OK* для окончания работы с мастером.

В итоге у вас создастся два объекта: oleDbDataAdapterl и oleDbConnectionl. Их предназначение аналогично тому, которое описано в главе «Работа с ADO.NET». Измените имена элементов на более читаемые: dataAdapter и connection, соответственно.

Код нашей программы будет представлен нижеприведенным листингом:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace OLEDBWinForms
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1: System.Windows.Forms.Form
  {
    private System.Data.OleDb.OleDbCommand oleDbSelectCommand1;
    private System.Data.OleDb.OleDbDataAdapter dataAdapter;
    private System.Data.OleDb.OleDbConnection connection;
    private System.Windows.Forms.ListBox listCustomers;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;
```

```
public Form1()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();

  //
  // TODO: Add any constructor code after InitiaiizeComponent call
  //
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
  if ( disposing )
  {
    if (components != null)
    {
      components.Dispose();
    }
  }
  base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitiaiizeComponent ()
{
  this.dataAdapter = new System.Data.OleDb.OleDbDataAdapter();
  this.oleDbSelectCommand1 = new System.Data.OleDb.OleDbCommand();
  this.connection = new System.Data.OleDb.OleDbConnection();
  this.listCustomers = new System.Windows.Forms.ListBox();
  //
  // dataAdapter
  //
  this.dataAdapter.SelectCommand = this.oleDbSelectCommand1;
  this.dataAdapter.TableMappings.AddRange(
    new System.Data.Common.DataTableMapping[]
    {
      new System.Data.Common.DataTableMapping(
```

```
        "Table",
        "Customers",
        new System.Data.Common.DataColumnMapping[]
        {
          new System.Data.Common.DataColumnMapping(
              "CompanyName", "CompanyName"),
          new System.Data.Common.DataColumnMapping(
              "ContactName", "ContactName")
        })
    });
//
// oleDbSelectCommand1
//
this.oleDbSelectCommand1.CommandText =
  "SELECT CompanyName, ContactName FROM Customers";
this.oleDbSelectCommand1.Connection = this.connection;
//
// connection
//
this.connection.ConnectionString =
  @"Provider=Microsoft.Jet.OLEDB.4.0;"+
  @"Password="""";User ID=Admin;"+
  @"Data Source=C:\northwind.mdb;"+
  @"Mode=Share Deny None;Extended Properties="""";"+
  @"Jet OLEDB:System database="""";"+
  @"Jet OLEDB:Registry Path=.....';"+
  @"Jet OLEDB:Database Password="""";"+
  @"Jet OLEDB:Engine Type=5;"+
  @"Jet OLEDB:Database Locking Mode=1;"+
  @"Jet OLEDB:Global Partial Bulk Ops=2;"+
  @"Jet OLEDB:Global Bulk Transactions=1;"+
  @"Jet OLEDB:New Database Password="""";"+
  @"Jet OLEDB:Create System Database=False;"+
  @"Jet OLEDB:Encrypt Database=False;"+
  @"Jet OLEDB:Don't Copy Locale on Compact=False;"+
  @"Jet OLEDB:Compact Without Replica Repair=False;"+
  @"Jet OLEDB:SFP=False";

// listCustomers
//
this.listCustomers.Location = new System.Drawing.Point(8, 8);
this.listCustomers.Name = "listCustomers";
this.listCustomers.Size = new System.Drawing.Size(280, 264);
this.listCustomers.TabIndex = 0;

//
```

```
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Name = "Form1";
        this.Text = "Form1";

    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
  }
}
```

Еще раз повторюсь, что в C# работа с базами данных с использованием объектов OLE DB очень похожа на работу с объектами SQL Server. Поэтому я но стану детально рассматривать каждую строку кода программы. Отдельно отмечу лишь те строки, которые отличаются от предыдущего примера.

```
this.connection.ConnectionString =
  @"Provider=Microsoft.Jet.OLEDB.4.0;"+
  @"Password="""";User ID=Admin;"+
  @"Data Source=C:\northwind.mdb;"+
  @"Mode=Share Deny None;Extended Properties="""";"+
  @"Jet OLEDB:System database="""";"+
  @"Jet OLEDB:Registry Path="""";"+
  @"Jet OLEDB:Database Password="""";"+
  @"Jet OLEDB:Engine Type=5;"+
  @"Jet OLEDB:Database Locking Mode=1;"+
  @"Jet OLEDB:Global Partial Bulk Ops=2;"+
  @"Jet OLEDB:Global Bulk Transactions=1;"+
  @"Jet OLEDB:New Database Password="""";"+
  @"Jet OLEDB:Create System Database=False;"+
  @"Jet OLEDB:Encrypt Database=False;"+
  @"Jet OLEDB:Don't Copy Locale on Compact=False;"+
  @"Jet OLEDB:Compact Without Replica Repair=False;"+
  @"Jet OLEDB:SFP=False";
```

Эти строки задают настройки для соединения с источником данных. Они определяют источник данных, пользователя, пароль, режимы работы клиента с базой данных и многое другое. Полный формат строки вы

сможете найти в MSDN. Нашей программе необходимы лишь четыре параметра:

```
Provider=Microsoft.Jet.OLEDB.4.0
```

Параметр Provider определяет тип источника данных. В нашем случае это «Microsoft.Jet.OLEDB.4.0», который позволяет работать с данными в формате Microsoft Access.

```
User ID=Admin
Password=""""
```

Параметры User ID и Password определяют имя пользователя для доступа к данным и его пароль. По умолчанию, в Microsoft Access используется имя Admin с пустым паролем.

```
Data Source=C:\northwind.mdb
```

Параметр Data Source определяет для Microsoft.Jet провайдера путь на диске к файлу базы данных. В моем случае это путь C:\northwind.mdb, у вас возможны другие варианты.

Расширение функциональности программы

Используя описание предыдущего примера, добавьте обработчик загрузки формы Form1_Load и код этого обработчика:

```
private void Form1_Load(object sender, System.EventArgs e)
    {
      //создаем новый объект DataSet
      DataSet ds = new DataSet();

      //вызываем функцию Fill для заполнения объекта
      //DataSet содержимым таблицы Customers
      dataAdapter.Fill(ds, "Customers");

      //получаем доступ к таблице Customers
      DataTable dt = ds.Tables["Customers"];

      //переносим данные в список
      //"Контактное имя" работает на "организация"
      //например Иван Петров работает на "СП СибирьНефть"
      foreach (DataRow dataRow in dt.Rows)
      {
        listCustomers.Items.Add( dataRow["ContactName"] t
          " работает на \" + dataRow["CompanyName"] + "\"");
      }
    }
```

Как вы могли заметить, в теле метода Form1_Load мы не исправили ни одной строки кода. Такая совместимость различных объектов ADO.NET дает возможность создавать легко переносимый код. То есть, если в один прекрасный момент вам придется отказаться от базы Microsoft Access в пользу, скажем, ORACLE, то у вас не возникнет больших проблем.

Запустив приложение, вы получите окно, изображенное на рис. 33.6. Результат остался прежним (ведь мы изменили только способ работы с данными, что никак не влияет на результаты выполнения программы).

ИСПОЛЬЗОВАНИЕ DataGrid

Возможности DataGrid

C# включает такой мощный инструмент для отображения данных, как элемент управления DataGrid. DataGrid может применяться как для разработки Windows Forms приложений, так и для разработки Web-приложений. Нет никакой разницы в написании кода.

Если вы имеете опыт работы с Windows, то должны хорошо представлять, что такое DataGrid. Этот элемент управления сочетает в себе все возможности таблицы Excel, и даже больше. Типичный образец использования элемента DataGrid —табличное представление данных; например, как представлено на рис. 33.9.

Создание примера приложения

Надо отметить, что наделение DataGrid мощными функциональными возможностями имеет и свои недостатки, одним из которых является необходимость использования дополнительных компонент. Сейчас мы рассмотрим работу с DataGrid на примере. Создадим приложение, ис-

Рис. 33.9. Использование DataGrid для табличного представления данных

ID	ФИО	Дом.телефон	ICQ
1			
2	Архутич Александр	8-01643-22237	42704099
3			
4	Володько Дмитрий	8-216-5-51-01	
5			
6	Кульпанович Александр		114103424
7			
8	Лях Борис	8-0177331147	
9			
10	Никифоров Евгений	240-60-30	75374929
11			
12	Рубинов Алексей	223-18-19	100309910
13			
14	Татаринов Александр	8-02230-22222	
16			
16	Шмелев Виталий	268-16-17	103984181

пользующее базу данных Northwind и SQL server в качестве источника данных.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace C_Sharp_Examples
{
  public class DataGridUsingForm: System.Windows.Forms.Form
  {
    private System.ComponentModel.Container
      components;
    private System.Windows.Forms.DataGrid
      Cust.omerDG;

    public DataGridUsingForm ( )
    {
      InitializeComponent( );

      // объявляем строку для подключения к базе
      string connectionString = "server=(local)\\NetSDK;" r
        "Trustedonnection=yes; database=northwind";

      //объявляем строку SQL запроса для выборки данных из базы
      string commandString =
        "Select CompanyName, ContactName, ContactTitle, "
        + "Phone, Fax from Customers";

      // создаем объект DataAdapter
      //задаем ему строку SQL запроса
      //задаем ему строку для соединения с базой
      SqlDataAdapter DataAdapter =
        new SqlDataAdapter(commandString, connectionString);

      //создаем объект DataSet
      DataSet ds = new DataSet();

      //заполняем таблицу "Customers" объекта ds данными из базы
      DataAdapter.Fill(ds, "Customers");

      // связываем источник данных объекта customerDG с
      // таблицей "Customers" объекта ds
      customerDG.DataSource=
        ds.Tables["Customers"].DefaultView;
    i
```

```
    protected override void Dispose(bool disposing)
    {
      if (disposing)
      {
        if (components == null)
        {
          components.Dispose();
        }
      }
      base.Dispose(disposing);
    }

    private void InitializeComponent( )
    {
      this.components =
        new System.ComponentModel.Container ( );
      this.customerDG =
        new System.Windows.Forms.DataGrid ( );
      customerDG.BeginInit ( );
      customerDG.Location =
        new System.Drawing.Point (8, 24);
      customerDG.Size =
        new System.Drawing.Size (656, 224);
      customerDG.DataMember = "";
      customerDG.TabIndex = 0;
      this.Text = "Using the Data Grid";
      this.AutoScaleBaseSize =
        new System.Drawing.Size (5, 13);
      this.ClientSize = new System.Drawing.Size (672, 273);
      this.Controls.Add (this.customerDG);
      customerDG.EndInit ( );
    }

    public static void Main(string[ ] args)
    {
      Application.Run (new DataGridUsingForm ( ) );
    }
  }
}
```

Анализ кода программы

Единственный код данной программы, который требует дополнительного объяснения (при условии, что вы внимательно прочитали предыдущие разделы), это связывание DataGrid и DataSet.

```
customerDG.DataSource=
  ds.Tables["Customers"].DefaultView;
```

Класс DataGrid имеет свойство DataSource, которое определяет источник данных для отображения. Однако это не тот источник данных, который используется классом SqlDataAdapter. Источником данных для DataGrid может являться объект одного из типов:

- DataTable;
- DataView;
- DataSet;
- DataViewManager.

В нашем случае мы использовали DataView. Для получения объекта DataView применялась инструкция:

```
ds.Tables["Customers"].DefaultView;
```

то есть получали у объекта ds (DataSet) таблицу Customers как объект типа DataTable. Мы имеем право использовать название таблицы как индексатор, поскольку определили это при заполнении ds.

```
DataAdapter.Fill(ds, "Customers");
```

Объект DataTable имеет свойство DefaultView, которое и применяется программой как источник данных для DataGrid.

Работа с приложением

Откомпилируйте и запустите программу. Вы получите окно, изображенное на рис. 33.10.

Обратите внимание, что программа заполнила заголовок таблицы названиями полей базы данных. Это делается автоматически. Кроме того, представленный список может сортироваться. Для проверки сортировки щелкните левой кнопкой мыши по одному из заголовков списка. Например, если вы используете для сортировки поле *CompanyName,* то записи таблицы будут рассортированы в соответствии с алфавитным принципом сравнения строк (a, b, c...). Окно с отсортированным списком представлено на рис. 33.11.

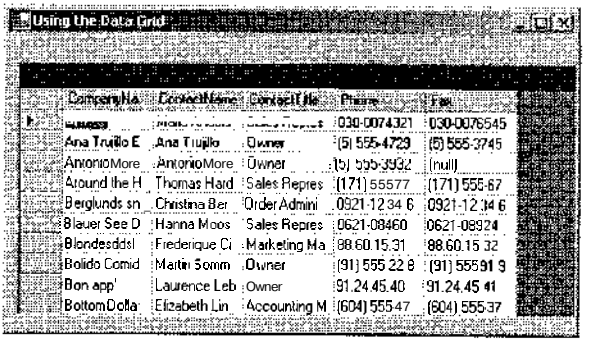


Рис. 33.10. Окно программы, использующей DataGrid

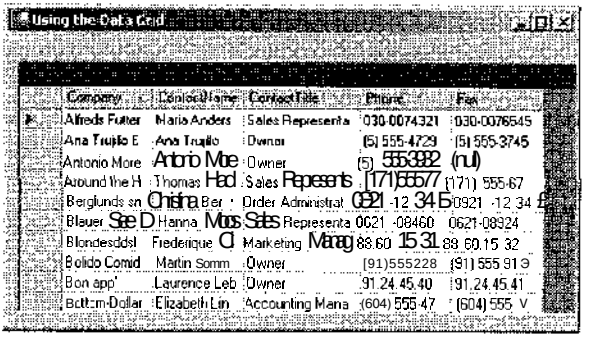


Рис. 33.11. Окно с отсортированным списком данных

Детальная настройка DataSet

Код, который мы выбрали для создания приложения в предыдущем разделе, был максимально оптимизирован и не давал полного представления о свойствах и методах используемых элементов. Однако существует возможность написания более понятного кода с детальной настройкой всех используемых элементов.

Для создания объекта DataAdapter использовался конструктор типа:

```
SqlDataAdapter DataAdapter =
  new SqlDataAdapter(commandString, connectionString);
```

Данный конструктор самостоятельно создает и инициализирует объекты SelectCommand и SqlConnection. Для создания управляемого кода программы лучше применять отдельное создание свойств объекта с последующим присвоением их объекту назначения.

```
private System. Data. SqlClient. SqlConnection myConnection;
private System.Data.DataSet myDataSet;
private System.Data.SqlClient.SqlCommand myCommand;
private System.Data.SqlClient.SqlDataAdapter DataAdapter;
```

Объявляем строку для подключения к базе.

```
string connectionString = "server-(local)\\NetSDK;" +
  "Trustedonnection-yes;database-northwind";
```

Создаем объект SqlConnection и инициализируем его строкой для подключения к базе.

```
rr.yConnection = new SqlConnection() ;
myConnection.ConnectionString = connectionString;
```

Затем явно открываем соединение с базой.

```
myConnection.Open ();
```

Явное открытие соединения с базой позволяет сократить время доступа к данным за счет использования одного и того же соединения в пределах одной сессии (одного сеанса работы с базой). Если вам необходимо подключиться к базе с иными правами доступа или другими опциями, тогда закройте соединение, инициализируйте заново строку для подключения и откройте соединение заново. ВСЕГДА ЗАКРЫВАЙТЕ СОЕДИНЕНИЕ! Особенно это касается создания серверных приложений. При уничтожении объекта SqlConnection соединение закрывается автоматически. Поэтому если ваша программа всегда будет работать с базой в пределах одной сессии, вы можете сделать исключение и не закрывать соединение. Однако это плохое правило. Я позволю себе опустить код закрытия соединения с базой, чтобы не перегружать программу лишними строками кода.

Затем создаем объект DataSet.

```
myDataSet = new DataSet(|;
```

Создаем объект SqlCommand и настраиваем его строкой выбора данных и объектом SqlConnection для подключения к базе. Строка выбора данных немного изменилась. Теперь мы получаем все поля таблицы Customers.

```
myCommand = new SqlCommand();
myCommand.Connection = myConnection;
myCommand.CommandText = "Select * From Customers";
```

Создание объекта DataAdapter тоже отличается от предыдущего примера. Мы явно указываем для этого объекта командную строку, явно настраиваем имена таблиц и явно заполняем DataSet.

```
DataAdapter = new SqlDataAdapter();
DataAdapter.SelectCommand = myCommand;
DataAdapter.TableMappings.Add("Table", "Customers");
DataAdapter.Fill(myDataSet);
```

Настройка элемента DataGrid не отличается от предыдущего примера.

```
customerDG.DataSource =
  myDataSet.Tables["Customers"].DefaultView;
```

Вот полный листинг программы с явной детальной настройкой компонент работы с базой данных.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace C_Sharp_Examples
{
  public class DataGridUsingForm: System.Windows.Forms.Form
  {
    private System.ComponentModel.Container
      components;

    private System.Data.SqlClient.SqlConnection myConnection;
    private System.Data.DataSet myDataSet;
    private System.Data.SqlClient.SqlCommand myCommand;
    private System.Data.SqlClient.SqlDataAdapter DataAdapter;

    private System.Windows.Forms.DataGrid
      customerDG;

    public DataGridUsingForm ( )
    {
      InitializeComponent ( ) ;

      // объявляем строку для подключения к базе
      string connectionString = "server=(local)\\NetSDK;" +
        "Trustedonnection=yes; database=northwind";

      // создаем и инициализируем объект Connection
```

```
    myConnection - new SqlConnection O ;
    myConnection.ConnectionString = connectionString;

    // явно открываем соединение
    myConnection.Open();

    //создаем объект DataSet
    myDataSet = new DataSet();

    //создаем объект SqlCommand
    myCommand = new SqlCommand();
    //инициализируем свойство Connection
    myCommand.Connection = myConnection;
    //инициализируем свойство CommandText
    rcyCommand.CommandText = "Select * From Customers";


    // создаем объект DataAdapter
    DataAdapter = new SqlDataAdapter();
    //задаем SQL команду
    DataAdapter.SelectCommand = myCommand;
    //задаем имя отображения таблицы
    DataAdapter.TableMappings.Add("Table", "Customers");
    //заполняем myDataSet данными из базы
    DataAdapter.Fill(myDataSet);

    // связываем источник данных объекта customerDG с
    // таблицей "Customers" объекта ds
    customerDG.DataSource =
      myDataSet.Tables["Customers"].DefaultView;
}

protected override void Dispose(bool disposing)
{
  if (disposing)
  {
    if (components == null)
    {
      components.Dispose();
    i
  ")
  base.Dispose(disposing);
}

private void InitializeComponent( )
i
```

```
      this.components =
        new System.ComponentModel.Container ();
      this.customerDG =
        new System.Windows.Forms.DataGrid ();
      customerDG.BeginInit ();
      customerDG.Location =
        new System.Drawing.Point (8, 24);
      customerDG.Size =
        new System.Drawing.Size (656, 224);
      customerDG.DataMember = "";
      customerDG.TabIndex = 0;
      this.Text = "Using the Data Grid";
      this.AutoScaleBaseSize =
        new System.Drawing.Size (5, 13);
      this.ClientSize = new System.Drawing.Size (672, 273);
      this.Controls.Add (this.customerDG);
      customerDG.EndInit ();
    }

    public static void Main(string[] args)
    {
      Application.Run(new DataGridUsingForm ());
    }
  }
}
```

На этот раз окно приложения будет выглядеть немного иначе. DataGrid будет содержать больше полей, чем все предыдущие примеры, использующие базу данных Northwind (см. рис. 33.12).

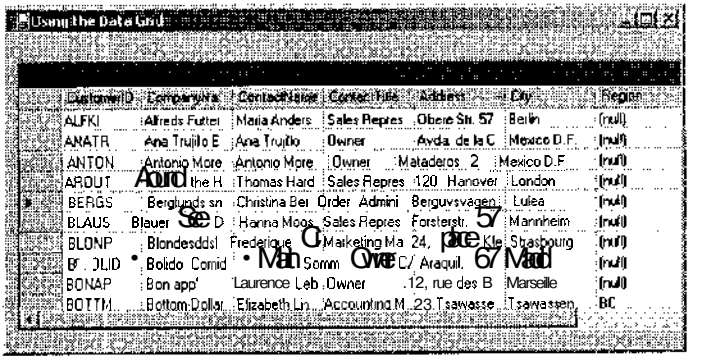


Рис. 33.12. Полный список записей таблицы Customers

34. ОТЛАДКА ПРОГРАММ

Если вы написали программу, и она заработала у вас с первого раза и правильно, считайте, что вам повезло. Вы можете считать себя хорошим программистом, если все ваши программы функционируют правильно сразу после завершения работы над кодом программы. Однако не всегда обстоятельства складываются так хорошо. Программирование — это тяжелый и кропотливый труд, который требует большого опыта и определенного склада ума. Поэтому, если первая попытка написать полностью функционирующий код вам не удалась, не расстраивайтесь, все можно поправить.

Если ваша программа проста в применении и все функции приложения визуально доступны, то все допущенные ошибки будут видны сразу после запуска программы.

Давайте рассмотрим программу, созданную как консольное приложение, которая меняет местами значения двух переменных.

```
using System;

namespace C_Sharprogramming
{
 class Exchange
 {
 public static void Main()
 {
 int a = 5;
 int b = 8;

 a = b;
 b = a;

 Console.WriteLine("a={0}, b={1}", a, b);
 }
 }
}
```

После запуска приложения на экране появится результат:

a=8, b=8

Программа явно работает неправильно. Естественно, многие сразу сообразят, в чем здесь ошибка, и с легкостью устранят ее. Однако давайте будем последовательны и воспользуемся программным отладчиком для поиска ошибки.

Для этого нам необходимо запустить программу в пошаговом режиме.

ПОШАГОВЫЙ РЕЖИМ

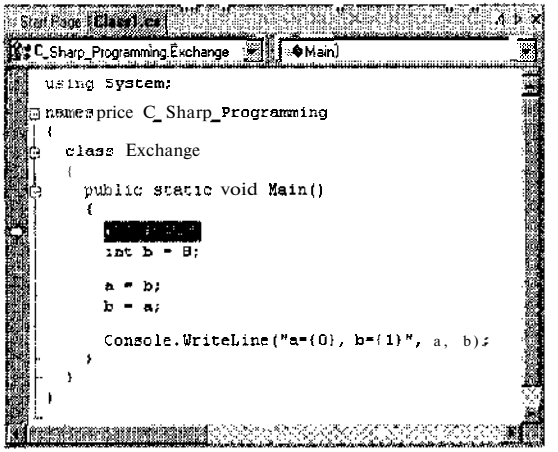


Рис. 34.1. Окно трассировки приложения

Пошаговый режим представляет собой процесс исполнения программы, при котором за один раз исполняется только одна инструкция. Для реализации этого режима нажмите клавишу F10 или выберите пункт меню *Debug/Step Over*. Обратите внимание, что первая инструкция в теле функции Main окажется выделенной желтым цветом (рис. 34.1). Тем самым указано место, с которого начнется выполнение программы. Также следует обратить внимание, что строки объявления используемых модулей, классов и переменных обходятся отладчиком, поскольку с ними не связаны действия, которые могут быть трассированы.

Нажмите клавишу F10 несколько раз и обратите внимание, как движется выделенная строка от одной к другой.

При пошаговом режиме вы можете заходить в вызываемые функции, используя клавишу F11 или пункт меню *Debug/Step Into*. В некоторых случаях не обязательно заходить в функции — достаточно только следить за их выполнением, для чего понадобится клавиша F10. Всякий раз при нажатии этой клавиши выполняется следующая инструкция, но трассировка вызова функции не происходит. Однако клавиша F11 не даст вам возможности для трассировки встроенных функций.

ТОЧКИ ОСТАНОВА

Как ни полезен пошаговый режим, в больших программах его использование может быть очень утомительным, особенно если отлаживаемый участок кода расположен глубоко в программе. Вместо клавиш F10 и F11 для достижения отлаживаемого участка кода гораздо удобнее использовать точки останова в начале критического участка кода, то есть того участка, который необходимо отладить. Точка останова, в соответствии со своим названием, означает остановку выполнения программы. Когда выполнение программы достигает точки останова, программа прекращает выполняться до того, как соответствующая строка кода будет выполнена. Управление возвращается отладчику, что позволяет проверить значение определенных переменных и начать режим пошаговой отладки.

Установить точку останова можно на любую инструкцию исполняемого кода. После задания одной или нескольких точек останова следует запустить программу при помощи клавиши F5 или меню *Debug/Start*. Существует два основных типа точек останова: условные и безусловные.

Безусловные точки останова

Безусловные точки останова всегда прекращают выполнение, как только встречаются в программе. Существует несколько способов добавить точки останова этого типа. Первый заключается в том, чтобы поместить курсор на той строке кода, где надо установить точку останова. Затем можно выбрать один из следующих способов задания точки останова:

— нажать клавишу F9;
— выбрать пункт меню *Debug/New Breakpoint* или **Ctrl+B**.

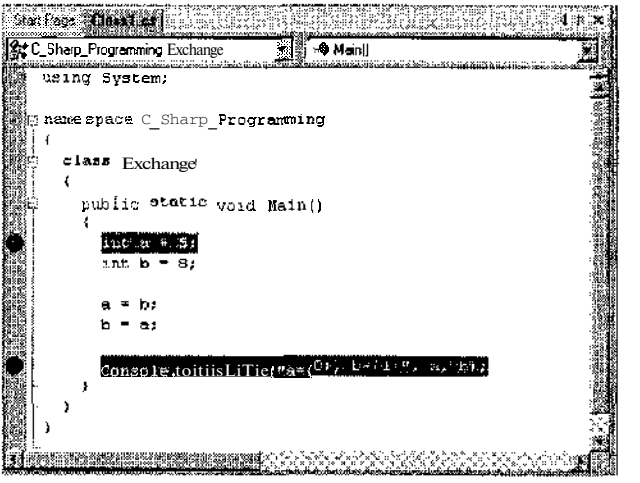


Рис. 34.2. Подсветка точек останова в программе

Если вы воспользуетесь клавишей F9, то та строка, где установлена точка останова, будет выделена цветом. В программе может быть несколько активных точек останова (рис. 34.2).

Если же вы выберете пункт меню *Debug/New Breakpoint,* то на экране появится окно, изображенное на рис. 34.3.

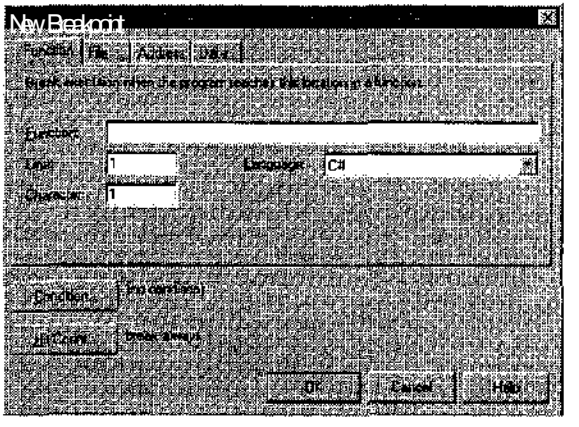


Рис. 34.3. Окно установки точки останова. Закладка *Function*

Закладка *Function* этого окна предоставляет возможность установить точку останова на любую функцию в программе. Для этого в поле *Function:* вам следует ввести имя функции, при вызове которой программа должна остановиться. При необходимости вы можете установить смещение точки останова в функции, указав значения в полях *Line* и *Character*. Закладка *File* окна *New Breakpoint* позволяет установить точку останова на любое место в любом файле проекта (рис. 34.4).

Для этого следует указать в поле *File* имя файла проекта, в который нужно поместить точку останова, а в полях *Line* и *Character* — указать номер строки и позицию символа в строке, на котором устанавливается точка останова.

Закладки *Address* и *Data* позволяют установить точки останова на конкретный адрес в программе или переменную, соответственно. Их обсуждение выходит за рамки этой книги, поэтому за более подробной информацией вы можете обратиться к MSDN.

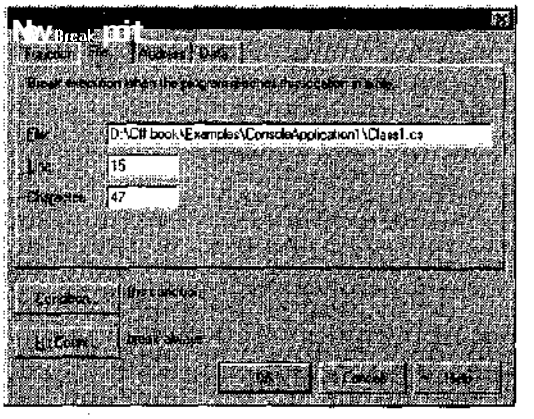


Рис. 34.4. Окно установки точки останова. Закладка *File*

Условные точки останова

Условные точки останова в коде программы позволяют указать условия, при которых точки останова прекратят выполнение программы, а также определить действия, которые произойдут при этом. Для прикрепления к точке останова определенного условия необходимо воспользоваться уже знакомым вам окном, изображенным на рис. 34.4. Давайте рассмотрим пример новой программы.

```
using System;

namespace C_Sharp_Programming
{
  class Exchange
  {
    public static void Main()
    {
      int a = 0;
      int b = 100;

      for(int i = 0; i < 100; i++)
      {
        a++;

        b--;

        Console.WriteLine("a={0}, b={1}", a, b);
      }
    }
  }
}
```

Как вы могли заметить, программа содержит цикл, повторяющийся 100 раз. Допустим, вам необходимо отследить ситуацию, когда значения

переменных a и b совпадают. Заранее скажу, что это произойдет на числе 50. Если следовать уже имеющимся у нас навыкам, то для этого необходимо установить точку останова на строку

```
Console.WriteLine("a={0}, b={1}", a, b);
```

и просматривать значения переменных a и b после каждой итерации. При этом вам придется 50 раз запускать программу после срабатывания точки останова. Попробуйте установить точку останова на указанной строке и достичь в цикле равенства переменных a и b. А что если вместо 100 поставить значение 1 000 000? Сколько времени вам на это понадобится? Упростить отладку такой программы помогает наложение условий на точки останова. Для установки условия в окне *New Breakpoint* существует кнопка *Condition...* При нажатии этой кнопки на экране появится окно, изображенное на рис. 34.5. Для нашего примера введите в поле *Condition* строку условия, как будто описываете инструкцию внутри блока **if**:

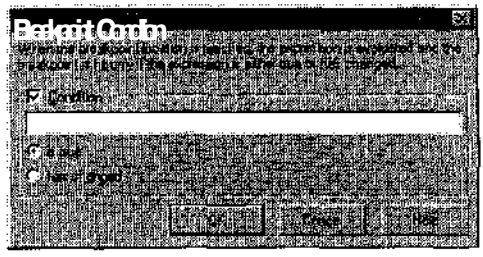


Рис. 34.5 Set Conditions

```
a == b
```

Установите флажок *is true* и нажмите OK. Запустите программу при помощи клавиши F5. В консоль выведется 49 строк, пока программа не остановится. Когда значения переменных a и b сравняются и станут равными 50, выполнение программы прервется и перейдет в окно отладчика (рис. 34.6).

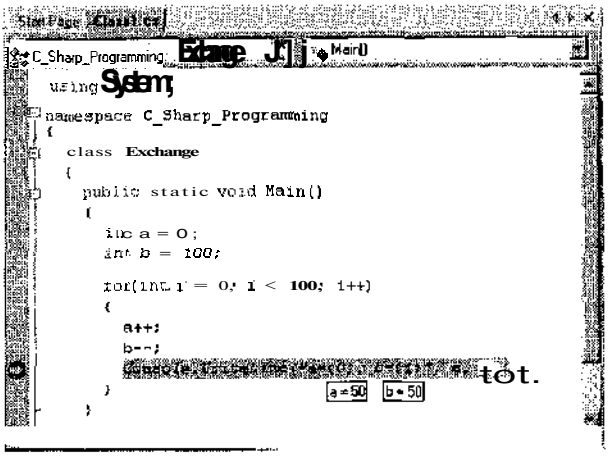


Рис. 34.6. Окно отладчика при срабатывании условия точки останова

Кроме наложения условия на точку останова, можно установить точку останова таким образом, чтобы она срабатывала после выполнения нескольких итераций цикла. Для этого необходимо воспользоваться кнопкой *Hit Count...,* расположенной в окне *New Breakpoint* (рис. 34.4). Перед вами появится окно, изображенное на рис. 34.7.

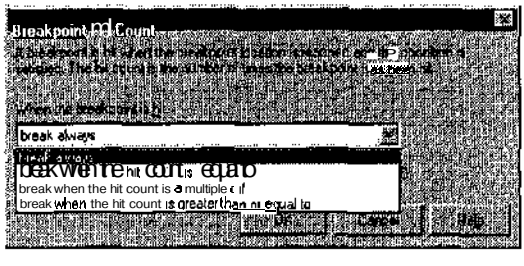


Рис. 34.7. Окно настройки количества повторений цикла

По умолчанию, в списке выбора типа повторений установлено значение *break always* (всегда). Это означает, что программа будет останавливаться на каждой итерации цикла. Если вы выберете пункт *break when the hit count is equal to,* то программа остановится на точке останова после выполнения указанного числа итераций. Если вы выберете пункт *break when the hit count is a multiple of,* то программа будет останавливаться лишь на тех итерациях, значение которых без остатка делится на указанное. Так, если вы укажете число 2, то программа будет останавливаться на каждой четной итерации цикла, если же укажете число 10, то программа будет останавливаться на каждой 10 итерации цикла. Последним пунктом списка является элемент *break when the hit count is greater than or equal to.* Выбор этого пункта заставит программу останавливаться на точке останова, если количество итераций цикла будет равным или превысит указанное значение.

Рассмотрим все тот же пример. Выберите в представленном списке пункт *break when the hit count is equal to.* Установите в поле для ввода значение 50. Запустите программу при помощи клавиши F5. Программа прервет свое выполнение на 50-й итерации.

ПРОСМОТР ПЕРЕМЕННЫХ

Во время отладки обычно необходимо видеть значения переменных программы. Visual Studio.NET позволяет легко справиться с этой задачей. Одна из возможностей просмотра переменных — это всплывающая подсказка, которая появляется при наведении курсора мыши на название переменной в коде программы.

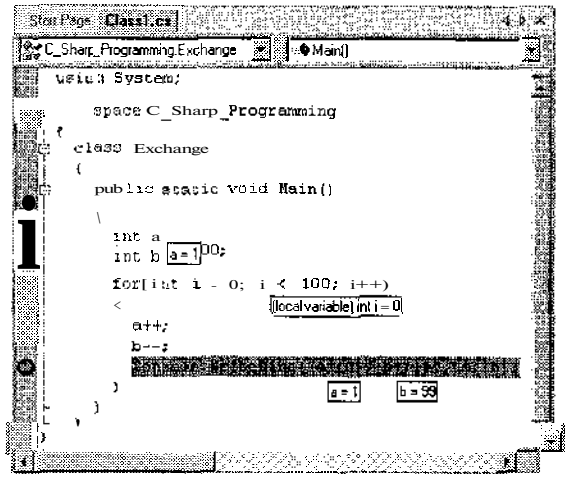


Рис. 34.8. Просмотр значений переменных в рабочей области программы

Воспользуемся все тем же примером. Вы можете установить точку останова на строку

```
Console.WriteLine("a={0}, b={1}",
a, b);
```

и запустить программу при помощи клавиши F5. Программа остановит свою работу, как только достигнет точки останова, и вы сможете увидеть значения переменных. Для этого подведите указатель мыши к переменной a в любом месте кода программы. При этом появится подсказка с указанием имени переменной ее типа и значением (см. рис. 34.8).

Второй, более обширной, возможностью просмотра значений переменных, является использование специального окна просмотра переменных (рис. 34.9).

По умолчанию, это окно содержит три закладки:

- Autos;
- Locals;
- Watchl.

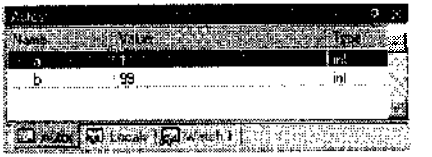


Рис. 34.9. Окно просмотра переменных

Закладка *Autos* содержит все переменные, которые модифицировались в предыдущем действии или будут модифицироваться в следующем. В предыдущем примере такими переменными являются переменные a и b. При переходе от одного участка кода к другому содержимое окна *Autos* автоматически обновляется в зависимости от производимого действия в трассируемой области.

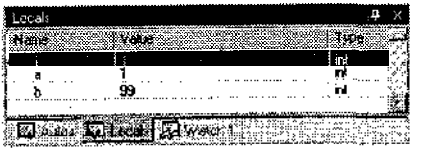


Рис. 34.10. Окно просмотра локальных переменных

Закладка *Locals* содержит все локальные переменные, видимые в трассируемой области программы. Для нашего случая локальными переменными являются a, b, i (рис. 34.10).

Закладка *Watchl,* по умолчанию, пуста. Она предназначена для просмотра переменных, определяемых программистом. Для просмотра значения необходимой переменной установите курсор мыши в поле *Name* выделенной строки и введите имя переменной. Ее значение сразу же отобразится в поле *Value.* Кроме того, в поле *Name* вы можете задавать любые выражения, содержащие видимые переменные, константы, вызовы функций. Например, так, как изображено на рис. 34.11.

Все вышеперечисленные окна позволяют изменять значения представленных в них переменных. Если вы измените значение переменной в окне просмотра, то изменится и конкретное значение переменной в памяти. Для изменения значения переменной необходимо щелкнуть указателем мыши по значению переменной в поле *Value.* При этом поле перейдет в режим редактирования и вы сможете модифицировать значение переменной.

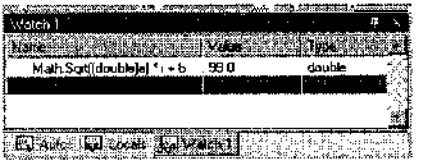


Рис. 34.11. Вычисление сложного выражения в окне Watchl

СТЕК ВЫЗОВА ФУНКЦИЙ

Очень часто необходимо отладить участок кода в функции, вызов которой предваряется вызовом многих других функций. Давайте рассмотрим следующий пример:

```
using System;

namespace C_Sharp_Programming
```

```
{
  class Test
  {
    public void f1()
    {
      f2();
    }
    public void f2()
    {
      Console.WriteLine("f2 выполняется");
    }
  }
  class TestApp
  {
    public static void Main()
    {
      Test test = new Test();
      test.f1();
    }
  }
}
```

Здесь функция Main класса TestApp вызывает функцию f1 класса Test. В свою очередь функция fl вызывает функцию *fl.* В больших программах вложенность функций будет намного выше. Бывает очень полезно посмотреть, какая именно функция вызвала ту функцию, которую вы трассируете. Для этого Visual Studio .NET содержит окно стека вызова функций *{Call Stack).* Это окно содержит упорядоченный, по очередности вызова, список функций. Установите в приведенном выше примере точку останова на строку

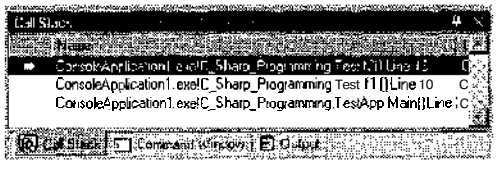


Рис. 34.12. Стек вызова функций

```
Console.WriteLine("f2 выполняется");
```

внутри функции f2. Запустите программу при помощи клавиши F5. Когда программа остановится на точке останова, взгляните на окно *Call Stack* внизу среды разработки Visual Studio .NET. Оно должно иметь вид, представленный на рис. 34.12.

Список содержит названия вызванных функций, записанных в обратном порядке. Первой была вызвана функция TestApp.Main, которая вызвала функцию Tets.fl, та, в свою очередь, вызвала функцию Test.f2, в которой сейчас и находится программа. Вы можете перейти назад по стеку вызова функций, щелкнув два раза указателем мыши по имени функции в списке. При переходе по стеку вызову функций изменяются и окна автоматических и локальных переменных.

ТАК ЧТО ЖЕ ЛУЧШЕ, C# ИЛИ Java?

Microsoft описывает C# как «простой, современный объектно-ориентированный и типо-безопасный язык программирования, наследник C и C++». Такое утверждение не хуже подходит и к Java. Как показывает следующий пример, возможности и синтаксис языков похожи.

Пример Hello World! на C#:

```
public class HelloWorldTest
{
 public static void Main (string[] args)
 {
 System.Console.WriteLine("Здравствуй, мир!");
 }
}
```

Аналогичный пример на Java:

```
public class HelloWorldTest
{
 public static void main (String[] args)
 {
 System.Out.println("Здравствуй, мир!");
 }
}
```

Но сходство идет дальше синтаксиса, ключевых слов и разделителей. Оно включает такие общие возможности Java и C#, как:

- Автоматическая сборка мусора (garbage collection),
- Механизм отражения (Reflection) для описания информации о типах.
- Компиляция исходного кода в промежуточный байт-код.
- ЛТ-компиляция байт-кода в собственно код.
- Исполнение кода в специальной защищенной среде.
- Все должно лежать в классе — никаких глобальных функций и данных.
- Отсутствие множественного наследования, хотя реализация нескольких интерфейсов и возможна.
- Дерево объектов с единым корнем.
- Специальные операторы для защиты кода и данных.
- Использование исключений для обработки ошибок.
- Ключевые слова package/namespace для предотвращения конфликтов типов.
- Соглашение о применении комментариев внутри кода в качестве документации.
- Проверка границ массивов.
- Встроенные библиотеки для работы с GUI, сетями, потоками и т. п.

- Отсутствие неинициализированных переменных. Если программист не инициализировал переменную, она автоматически инициализируется компилятором.
- По умолчанию, использование указателей запрещено.
- Использование package/namespace для ссылок на внешние библиотеки и компоненты вместо применения заголовочных файлов.
- Использование точки вместо операторов C++ «->» и «::».
- Ужесточение контроля типов. Так, в операторах if нельзя использовать целочисленные значения без явного приведения их к типам boolean/bool.

C#: ЭВОЛЮЦИЯ Visual J++

С чего Microsoft решил, что нам нужен новый язык? Microsoft вложил уйму сил и средств в проект Visual J++, объявленный в октябре 1996 года. Эти усилия произвели на свет самую быструю JVM на рынке и Windows Foundation Classes (WFC), набор Java-классов, оберток для Win32 API. Не случайно Андерс Хиджисберг, руководитель проекта WFC (более известный как автор Turbo Pascal), стал главным архитектором C#, ввиду чего C# немало унаследовал и от Turbo Pascal.

В Microsoft решили внести изменения в Java для более тесной интеграции с Windows. Вы уже знаете, почему. Некоторые изменения — бесшовное сопряжение с COM, отказ от поддержки RMI и JNI и введение делегатов — привели к нарушению совместимости со стандартом Java. Вследствие этого Sun Microsystems предъявило иск Microsoft в октябре 1997 года за нарушение лицензионного соглашения. Это был приговор будущему разработок Java и Visual J++ компании Microsoft. Но в Microsoft решили использовать наработки в Java, компиляторах и JVM и преобразовать их в еще более амбициозный проект — Microsoft .NET.

Программы, написанные на C#, компилируются в промежуточный язык под названием «MSIL», с некоторой натяжкой его можно назвать эквивалентом байт-кода или p-кода Visual Basic. Как уже говорилось ранее, любой язык, который можно скомпилировать во MSIL, может воспользоваться такими возможностями CLR, как сборка мусора, отражения, метаданные, контроль версий, события и защита. Кроме этого, класс, написанный на одном языке, может наследовать от класса на другом языке и подменять его методы.

Интересно, что Microsoft продвигает кросс-языковую разработку, а Sun/Java — кросс-платформную.

К сожалению, и у того, и у другого подхода есть свои недостатки. Создание компонент на нескольких языках всегда влечет за собой проблемы взаимодействия. Более того, всегда есть проблема — что делать, когда ваш программист-постановщик задач найдет местечко потеплее. Кросс-платформная разработка тоже никогда не была безупречна, как знают Java-программисты, особенно в части GUI и потоков.

СХОДСТВО C# И Java

Не говоря о множестве других общих черт, большинство ключевых слов в Java имеют соответствия в C#. Некоторые ключевые слова совпадают, например, new, bool, this, break, static, class, throw, virtual и null. Этого можно было ожидать, поскольку данные ключевые слова заимствованы из C++. Занимательно, что многие ключевые слова Java, не имеющие прямых эквивалентов в C#, например, super, import, package, synchronized и final, просто называются по-другому в C# (base, using, namespace, lock и sealed, соответственно). Ниже в таблице приведены некоторые ключевые слова и черты, существующие в обоих языках, но выглядящие слегка иначе в C#.

	Java	C#
Наследование	class Foo extends Bar implements IFooBar { }	class Foo: Bar, IFooBar { }
Ссылка на базовый класс	super.hashCode ();	base.GetHashCode ();
Использование внешних типов	import Java.util;	using System.Net;
Пространства имен	package MyStuff; class MyClass { }	namespace MyStuff { class MyClass {...} }
Предотвращение наследования	final class Foo {}	sealed class Foo {}
Константы	final static int MAX = 50;	const int MAX = 50;
Комментарии	/** * Convert string to * uppercase. * * @param s The string to * convert * * ©return The uppercase * string V	/// <summary> /// Convert string to /// uppercase. /// </summary> /// <param name="s"> /// The string to convert /// </param> /// <returns> /// The uppercase string /// </returns>
Устаревшие, не рекомендуемые к использованию классы или методы	©deprecated Не использовать!	[obsolete ("Не использовать!")]
Синхронизация	synchronized (this) { ++ refs; }	lock (this) { ++ refs; }

КЛАСС Object

Другой хороший пример косметических различий двух языков — класс System.Object в C#, имеющий точно те же методы, что и Java-класс Java.long.Object, если не считать того, что они несколько иначе пишутся. Метод clone в Java называется MemberwiseClone в C#, equals из Java — это Equals в C#, getClass — getType, hashCode — GetHashCode, а toString — ToString.

Простое совпадение? Да, в недавнем интервью Андерс Хиджисберг говорил, что «C# — это не клон Java». Совершенно верно, это MemberwiseClone.

МОДИФИКАТОРЫ ДОСТУПА

В отличие от C++, в C# модификатор public и ему подобные относятся к одному определению (метода, класса, переменной) наподобие того, как это сделано в Java. Модификаторы public и private имеют одно и то же значение во всех трех языках. Но «protected access» в Java называется «protected internal» в C#, а «package access» в Java называется «internal» в C#. Модификатор protected в C# дает наследникам доступ к членам класса, даже если наследники находятся в другой программе. Другое различие — в Java по умолчанию применяется package access, а в C# — private.

ЧТО В C# ЛУЧШЕ, ЧЕМ В Java

Итак, если C# и Java во многом выглядят и действуют похоже, зачем же вообще использовать C#? Как и следует ожидать от очередного языка, C# совершеннее Java в некоторых областях. В дополнение к появившимся новым возможностям, в C# проще синтаксис для таких вещей, как итерации, события и работа с примитивными типами вроде объектов и т. п., что уменьшает количество кода и ускоряет процесс разработки.

КОНТРОЛЬ ВЕРСИЙ

Контроль версий — очень нужная в Windows- и Java-разработке вещь. Java-разработчики не хуже остальных знакомы с устаревшими API и несовместимыми версиями сериализованных объектов.

.NET в очередной раз обещает устранить эти проблемы, позволяя указать зависимости версий между компонентами и поддержкой одновременного исполнения нескольких версий компонента. Более того, на ком-

пьютере может иметься несколько версий runtime .NET одновременно, что позволяет использовать в каждом случае ту версию, под которую компилировались компоненты. Это похоже на возможности контроля версий, оговоренные в Java Product Versioning Specification, за исключением того, что в .NET Framework принудительный контроль во время исполнения, компиляции и инсталляции уже встроен.

СРЕДСТВА ОТЛАДКИ ВО ВРЕМЯ ИСПОЛНЕНИЯ

Как уже говорилось, среди стандартных библиотек CLR присутствует библиотека, дающая возможности runtime-отладки — вывода сообщения в консоль, выдачи Assert и т. п. Естественно, C# предоставляет полный доступ к этим возможностям, как, впрочем, и любой другой CLR-совместимый язык.

Assert не поддерживаются в Java, но, как и над большинством других недоработок, в Java Community Process над этим работают.

ref- И out-ПАРАМЕТРЫ

В C# параметры можно передавать по значению (in), по ссылке (ref), или указывать, что это — выходные параметры (out). Так как Java позволяет только передачу по значению, для передачи в метод чего-либо по ссылке приходится заниматься натуральным шаманством, например, использовать возвращаемые значения, передачу в одноэлементном массиве или помещение объекта в класс-обертку. Метод, принимающий параметр по ссылке в C#:

```
void swap (ref long n1, ref long n2) ;
```

Вызов C#-метода:

```
int i1 = 123, i2 = 321;
swap (i1, i2);
```

Аналогичный пример на Java:

```
void swap (long[] n1, long[] n2);
```

Вызов java-метода:

```
int i1 = 123, i2 = 321;
int[1] itmp1;
itmp1 = i1;
int [1] itmp2;
itmp2 = i2;
swap (itmp1, itmp2);
```

Необходимо снова подчеркнуть, что возможность определения параметров методов как прямых, передающихся по ссылке или обратно возвращаемых, заложена на уровне CLR, а конкретные языки только предоставляют некоторый синтаксис, позволяющий воспользоваться этими возможностями.

ВИРТУАЛЬНЫЕ МЕТОДЫ

По умолчанию, в Java все методы виртуальны и могут подменяться наследниками класса. C# более привержен традициям. В нем, как и в C++, виртуальные методы должны быть явно объявлены виртуальными. Более того, C# несколько улучшает модель декларации виртуальных методов, заставляя программиста явно указывать, что он хочет подменить реализацию базового класса. Это позволяет избежать трудноуловимых ошибок, часто встречающихся в C++- и Java-программах. Например, стоит в C++ при подмене методов неумышленно изменить сигнатуру метода базового класса (описать параметр как long вместо int), и вместо ожидаемого переопределения метода вы создадите дополнительную (перегруженную) реализацию метода. При этом базовый класс будет вызывать свою реализацию. Программист может часами смотреть на место, где допущена ошибка, и не видеть ее. C# интерпретирует неявную перегрузку как ошибку компиляции, требуя от производного класса использовать ключевое слово override для подмены виртуального метода. Кроме этого, при компиляции выдается предупреждение, если метод производного класса скрывает метод базового класса. В этом случае, чтобы убрать предупреждение, можно использовать ключевое слово new.

ПЕРЕЧИСЛЕНИЯ (enums)

В отличие от C/C++, в Java не реализованы перечисления, так как создатели Java заявили, что перечисления не объектно-ориентированы. По правде говоря, enums куда безопаснее и удобнее, чем static final int в Java. Перечисления не только существуют в C#, они типо-безопасны, к ним можно применять операторы ++, -, <, и > и конвертировать в строку и обратно.

ТИП ДАННЫХ decimal

В C# поддерживается типы данных decimal, пришедший в CLR из Automation и Visual Basic. decimal — 128-битный тип, имеющий большую точность и меньший разброс, чем типы с плавающей точкой. Он особенно полезен для финансовых приложений.

ВЫРАЖЕНИЯ switch

Очень распространенной ошибкой C/C++-программистов, особенно начинающих, является пропущенный оператор break в конце раздела case оператора switch. Программист подсознательно считает, что в конце раздела case управление должно быть прервано и продолжиться после опе-

ратора switch. Но в C++ это не так, причем умышленно — чтобы позволить программисту создать один раздел с несколькими метками. В других языках обычно применяется перечисление меток через запятую в одном case-операторе. В 90 процентах случаев это удовлетворяет имеющимся потребностям, но реализация C++ более гибка. Разработчики C# нашли довольно красивое альтернативное решение.

По умолчанию при достижении конца case-раздела происходит выход из оператора switch. Чтобы достичь такой же функциональности, как в C++, можно применять оператор goto. В качестве меток используются соответствующие case-метки. Кроме того, с C# как метки можно использовать не только целые числа, но и строки (прямое заимствование из VB).

ДЕЛЕГАТЫ И СОБЫТИЯ

Делегаты — это объектно-ориентированные аналоги указателей на функции, предоставляющие типо-безопасный механизм для реализации функций обратного вызова и событий. Они могут указывать на статические методы или на нормальные методы класса. Эта возможность также является не особенностью языка, а унаследована от CLR, но именно в C# она наиболее органично встроена в язык.

Java работает с событиями через модель событий JavaBeans и классы-адаптеры. Обработка событий в C# проще и требует только реализовать индивидуальные методы вместо целых интерфейсов.

Как видно из этой главы, прямым предком C# был Visual J++. Именно в нем появились так называемые делегаты и сама концепция делегирования. Именно эта замечательная возможность стала каплей, переполнившей чашу терпения Sun Microsystems. Как вы думаете, в каком смертном грехе Sun обвинил делегаты? Правильно, делегаты не объектно-ориентированны. Ключевое слово event вводит в ваш класс делегата, позволяющего рассылать события.

ПРОСТЫЕ ТИПЫ (Value-ТИПЫ)

Как говорилось выше, все типы данных, поддерживаемые CLR, наследуются от базового класса object. Поэтому простые типы в C# (long, int, char) можно использовать везде, где необходима передача ссылки на объект. Java в данном случае проигрывает, поскольку простые типы в ней не рассматриваются как классы и в некоторых контекстах для работы с ростыми типами их приходится заворачивать в специальные классы-обертки, например, java.lang.Long. C#-компилятор неявно конвертирует простые типы в объекты (и наоборот) по потребности, через процесс под названием «упаковка и распаковка (boxing и un-boxing)». Причем если простые типы не рассматриваются как объекты, это не влечет никаких

накладных расходов. Это позволяет C#-разработчикам смотреть на мир как на целостную систему типов. Ниже показан пример использования простых типов как объектов на Java:

```
int i = 10;
System.out.println((new Integer(i)).hashCode()) ; // Используется класс-обертка
```

А вот так это выглядит на C#:

```
int i = 10;
System.Console.WriteLine(i.GetHashCode()); // Скрытая конвертация
```

СВОЙСТВА

Хотя во многих языках свойства и поля выглядят одинаково, реально свойства и поля — это две большие разницы. Поле — это член данных класса, а свойство — это пара методов получения и предоставления доступа, предоставляющих доступ или вычисляющих нужные значения.

И C#, и Java (посредством JavaBeans) поддерживают работу со свойствами. Однако в C# свойства напрямую поддерживаются языком вместо использования отражений и соглашений именования, как в JavaBeans. Это дает краткий, легко читаемый синтаксис. Приведенный ниже пример показывает реализацию свойства Name с использованием поля name на Java:

```
String name;
public String getName() { return name; }
public void setName(String value) { name = value; }
```

Использование свойства:

```
obj.setName ("Marc");
if (obj.getName() != "Marc")
 throw new Exception("Это не я!");
```

и на C#:

```
string name;
public string Name
{
 get { return name; }
 set { name = value; } // 'value' — это новое значение
1
```

Использование свойства:

```
obj.Name = "Marc";
if (obj.Name != "Marc")
 throw new System.Exception ("Это не я!");
```

Преимущества стиля C# в том, что методы get и set находятся в том же блоке кода и в том, что для обращения к свойствам используется более интуитивный синтаксис, поскольку доступ к свойствам производится точно так же, как к нормальным полям. Свойства JavaBeans тоже выглядят как нормальные поля, когда к ним обращаются из скриптов или визуальных дизайнеров, но не при доступе из Java-кода.

ИНДЕКСИРУЕМЫЕ СВОЙСТВА И СВОЙСТВА ПО УМОЛЧАНИЮ

Кроме переопределения операторов, в C# можно определять индексируемые свойства. К тому же, индексируемые свойства можно помечать как свойства по умолчанию для класса. Это позволяет рассматривать объект как массив, каждый элемент которого становится доступным с помощью операторов доступа к свойствам (get и set). Ниже приведен пример класса с индексированным свойством по умолчанию:

```
public class Skyscraper
{
 Story[] stories;
 public Story this [int index]
 {
 get { return stories [index]; }
 set
 {
 if (value != null)
 stories [index] = value;
 }
 }
...
}
```

Применение класса с индексированным свойством по умолчанию:

```
Skyscraper empireState = new Skyscraper (...);
empireState[102] = new Story ("The Top One",... );
```

МАССИВЫ, КОЛЛЕКЦИИ И ИТЕРАЦИИ

Выражение foreach в C# (украденное из VB) позволяет легко перечислять классы, поддерживающие интерфейс Enumerable, включающий массивы и коллекции. Это избавляет от необходимости писать for (int i=0; i < ary.length;i++) и триаду getIterator/hasNext/next как в Java и C++. Следующий листинг демонстрирует итерации на Java:

```
import java.util.*;
class Iterate
{
 public static void main (String [] args)
 {
 // Перебор массива аргументов командной строки
 for (int i = 0; i < args.length; i++)
 System.out.println(args [i]);

 // Создание связанного списка
```

```
 LinkedList list = new LinkedList();
 list.add ("Элемент 1");
 list.add ("Элемент 2") ;

 // Перебор значений списка
 Listlterator it = list.listlterator (0);
 while (it.hasNext())
 System.out.println (it.next());
 }
}
```

И на C#:

```
using System.Collections;
class Iterate
1
 public static void Main(string [] args]
  f
 // Перебор массива аргументов командной строки
 foreach (string arg in args)
 System.Console.WriteLine (arg);
 // Создание связанного списка
 ObjectList list = new ObjectList ();
 list.Add("Элемент 1");
 list.Add("Элемент 2");
 // Перебор значений списка
 foreach (string str in list)
 System.Console.WriteLine (str);
 }
}
```

ИНТЕРФЕЙСЫ

Интерфейсы C# похожи на интерфейсы Java, но C# дает большую гибкость при их использовании. Класс может реализовать методы интерфейса явно или неявно:

```
public interface ITeller
 (
 voidNext ();
}

public interface IIterator
 {
 voidNext ();
}
```

```
public class Clark: ITeller, IIterator
{
 void ITeller.Next () {... }
 void IIterator.Next () {... }
}
```

Это дает классу двойную выгоду. Во-первых, класс может реализовать несколько интерфейсов, не беспокоясь о конфликтах имен. Во-вторых, класс может «спрятать» метод, если он не должен быть доступен всем пользователям класса. Чтобы получить доступ к методам, при реализации которых было явно указано имя интерфейса, необходимо использовать явное приведение типов:

```
Clark clark = new Clark ();
((ITeller)clark).Next();
```

МНОГОМЕРНЫЕ МАССИВЫ

C# позволяет создавать как вложенные, так и многомерные массивы. Вложенные массивы очень похожи на реализацию массивов в C++ и Java. Многомерные массивы обеспечивают более точное и эффективное решение некоторых задач. Пример такого массива:

```
int [,,] array = new int [3, 4, 5]; // creates 1 array
int [1,1,1] = 5;
```

Использование вложенных массивов:

```
int [][][] array = new int [3][4][5]; // creates 1+3+12=16 arrays
int [1][1][1] = 5;
```

ПРИЛОЖЕНИЕ

Полный листинг программы «Графический редактор»

Файл Forml.cs

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace GraphEditorApp
{
  public enum Tools
  {
    PEN = 1, TEXT, LINE, ELLIPSE, NONE = 0
  }
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1: System.Windows.Forms.Form
  {
    public Tools currentTool;
    private System.Windows.Forms.ToolBar toolBar1;
    private System.Windows.Forms.ImageList imageList1;
    private System.Windows.Forms.ToolBarButton toolBarButtonPen;
    private System.Windows.Forms.ToolBarButton toolBarButtonLine;
    private System.Windows.Forms.ToolBarButton toolBarButtonEllipse;
    private System.Windows.Forms.MainMenu mainMenu1;
    private System.Windows.Forms.MenuItem menuItemFile;
    private System.Windows.Forms.MenuItem menuItemNew;
    private System.Windows.Forms.MenuItem menuItemClose;
    private System.Windows.Forms.MenuItem menuItemWindow;
    private System.Windows.Forms.MenuItem menuItemTool;
    private System.Windows.Forms.MenuItem menuItemPen;
    private System.Windows.Forms.MenuItem menuItemLine;
    private System.Windows.Forms.MenuItem menuItemEllipse;
    private System.Windows.Forms.ToolBarButton toolBarButtonText;
    private System.Windows.Forms.MenuItem menuItemText;
```

```
    private System.ComponentModel.IContainer components;

    public Form1 ()
    (
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if (components != null)
        {
          components.Dispose() ;
        )
    } 1
      base.Dispose( disposing );
    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do nor modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
      this.components = new System.ComponentModel.Container();
      System.Resources.ResourceManager resources = new
System.Resources.ResourceManager(typeof(Form1));
      this.toolBar1 - new System.Windows.Forms.ToolBar();
      this.toolBarButtonPen = new System.Windows.Forms.ToolBarButton ();
      this.toolBarButtonText = new System.Windows.Forms.ToolBarButton();
      this.toolBarButtonLine - new System.Windows.Forms.ToolBarButton();
      this.toolBarButtonEllipse= new System.Windows.Forms.ToolBarButton();
      this.imageList1 = new System.Windows.Forms.ImageList(this.components) ;
      this.mainMenu1 = new System.Windows.Forms.MainMenu();
```

```
        this.menuItemFile = new System.Windows.Forms.MenuItem();
        this.menuItemNew = new System.Windows.Forms.MenuItem();
        this.menuItemClose = new System.Windows.Forms.MenuItem();
        this.menuItemTool = new System.Windows.Forms.MenuItem();
        this.menuItemPen = new System.Windows.Forms.MenuItem();
        this.menuItemText = new System.Windows.Forms.MenuItem();
        this.menuItemLine = new System.Windows.Forms.MenuItem();
        this.menuItemEllipse = new System.Windows.Forms.MenuItem();
        this.menuItemWindow = new System.Windows.Forms.MenuItem();
        this.SuspendLayout();
        //
        // toolBar1
        //
        this.toolBar1.Buttons.AddRange(new System.Windows.Forms.ToolBarButton[] {


this.toolBarButtonPen,


this.toolBarButtonText,


this.tOOlBarButtonLine,


this.toolBarButtonEllipse});
        this.toolBar1.DropDownArrows = true;
        this.toolBar1.ImageList = this.imageList1;
        this.toolBar1.Name = "toolBar1";
        this.toolBar1.ShowToolTips = true;
        this.toolBar1.Size = new System.Drawing.Size(292, 25);
        this.toolBar1.TabIndex = 0;
        this.toolBar1.ButtonClick += new
System.Windows.Forms.ToolBarButtonClickEventHandler(this.toolBar1_ButtonClick);
        //
        // toolBarButtonPen
        //
        this.toolBarButtonPen.ImageIndex = 0;
        this.toolBarButtonPen.Style =
System.Windows.Forms.ToolBarButtonStyle.ToggleButton;
        //
        // toolBarButtonText
        //
        this.toolBarButtonText.ImageIndex = 1;
        this.toolBarButtonText.Style =
System.Windows.Forms.ToolBarButtonStyle.ToggleButton;
```

```
      //
      // toolBarButtonLine
      //
      this.toolBarButtonLine.ImageIndex = 2;
      this.toolBarButtonLine.Style =
System.Windows.Forms.ToolBarButtonStyle.ToggleButton;
      //
      // toolBarButtonEllipse
      //
      this.toolBarButtonEllipse.ImageIndex = 3;
      this.toolBarButtonEllipse.Style =
System.Windows.Forms.ToolBarButtonStyle.ToggleButton;
      //
      // imageList1
      //
      this.imageList1.ColorDepth = System.Windows.Forms.ColorDepth.Depth8Bit;
      this.imageList1.ImageSize = new System.Drawing.Size(16, 16);
      this.imageList1.ImageStream =
((System.Windows.Forms.ImageListStreamer)(resources.GetObject("imageList1.ImageStream")));
      this.imageList1.TransparentColor = System.Drawing.Color.Transparent;
      //
      // mainMenu1
      //
      this.mainMenu1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {


this.menuItemFile,


this.menuItemTool,


this.menuItemWindow});
      //
      // menuItemFile
      //
      this.menuItemFile.Index = 0;
      this.menuItemFile.MenuItems.AddRange(new System.Windows.Forms.MenuItem[]
{


 this.menuItemNew,


 this.menuItemClose});
      this.menuItemFile.Text = "&Файл";
```

```
      //
      //menuItemNew
      //
      this.menuItemNew.Index = 0;
      this.menuItemNew.Text = "&Создать";
      this.menuItemNew.Click += new
System.EventHandler(this.menuItemNew_Click);
      //
      // menuItemClose
      //
      this.menuItemClose.Index = 1;
      this.menuItemClose.Text = "&Закрыть";
      //
      // menuItemTool
      //
      this.menuItemTool.Index = 1;
      this.menuItemTool.MenuItems.AddRange(new System.Windows.Forms.MenuItem[]
{


 this.menuItemPen,


 this.menuItemText,


 this.menuItemLine,


 this.menuItemEllipse});
      this.menuItemTool.Text = "&Инструмент";
      //
      // menuItemPen
      //
      this.menuItemPen.Index = 0;
      this.menuItemPen.Text = "&Карандаш";
      this.menuItemPen.Click += new
System.EventHandler(this.menuItemTool_Click);
      //
      // menuItemText
      //
      this.menuItemText.Index = 1;
      this.menuItemText.Text = "&Текст";
      this.menuItemText.Click += new
System.EventHandler(this.menuItemTool_Click);
      //
```

```
        //menuItemLine
        //
        this.menuItemLine.Index = 2;
        this.menuItemLine.Text = "&Линия";
        this.menuItemLine.Click += new
System.EventHandler(this.menuItemTool_Click);
        //
        // menuItemEllipse
        //
        this.menuItemEllipse.Index = 3;
        this.menuItemEllipse.Text = "&Эллипс";
        this.menuItemEllipse.Click += new
System.EventHandler(this.menuItemTool_Click);
        //
        // menuItemWindow
        //
        this.menuItemWindow.Index = 2;
        this.menuItemWindow.MdiList = true;
        this.menuItemWindow.Text = "&Окно";
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Controls.AddRange(new System.Windows.Forms.Control[] {

                                              this.toolBar1});
        this.IsMdiContainer = true;
        this.Menu = this.mainMenu1;
        this.Name = "Form1";
        this.Text = "Графический редактор";
        this.ResumeLayout(false);

    }

    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }

    private void toolBar1_ButtonClick(object sender,
```

```
System.Windows.Forms.ToolBarButtonClickEventArgs e)
    {
      switch(e.Button.ImageIndex)
      {
        case 0:
          // установка режима рисования карандашом
          currentTool = Tools.PEN;
          break;
        case 1:
          // установка режима заливки
          currentTool = Tools.TEXT;
          break;
        case 2:
          // установка режима рисования линий
          currentTool = Tools.LINE;
          break;
        case 3:
          // установка режима рисования эллипсов
          currentTool = Tools.ELLIPSE;
          break;
      }

      SetToolBarButtonsPushedState(e.Button);

      SetMenuItemsCheckedState(menuItemTool.MenuItems[e.Button.ImageIndex]);
    }
    private void SetToolBarButtonsPushedState(ToolBarButton button)
    {
      foreach (ToolBarButton btnItem in toolBar1.Buttons)
      {
        if(btnItem == button)
        {
          btnItem.Pushed = true;
        }
        else
        {
          btnItem.Pushed = false;
        }
      }
    }

    private void menuItemTool_Click(object sender, System.EventArgs e)
    {
      // получаем пункт меню
      MenuItem item = (MenuItem)sender;
```

```
    switch(item.Text)
    \
      case "&Карандаш":
        // установка режима рисования карандашом
        currentTool = Tools.PEN;
        break;
      case "&Заливка":
        // установка режима заливки
        currentTool = Tools.TEXT;
        break;
      case "&Линия":
        // установка режима рисования линий
        currentTool = Tools.LINE;
        break;
      case "&Эллипс":
        // установка режима рисования эллипсов
        currentTool = Tools.ELLIPSE;
        break;
    }

    // устанавливаем состояние пунктов меню
    SetMenuItemsCheckedState(item);

    // устанавливаем состояние кнопок
    SetToolBarButtonsPushedState(toolBar1.Buttons[item.Index]);
  }

  private void SetMenuItemsCheckedState(MenuItem item)
  {
    // для каждого пункта меню
    foreach(MenuItem raenultem in raenuItemTool.Menulteras)
    {
      if(menuItem == item)
      {
        menuItem.Checked = true;
      }
      else
      {
        menuItem.Checked - false;
      }
    }
  }

  private void menuItemNew_Click(object sender, System.EventArgs e)
  {
    Form2 newMDIChild = new Form2();
```

```
        newMDIChild.MdiParent = this;

        newMDIChild.Show();
      }
    }
}
```

Файл Form2.cs

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace GraphEditorApp
{
  /// <summary>
  /// Summary description for Form2.
  /// </summary>
  public class Form2: System.Windows.Forms.Form
  {
    public bool drawPen;
    public bool FirstClick;
    public Point PreviousPoint;

    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;


    public Form2()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
```

```
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if(components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
      //
      // Form2
      //
      this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
      this.BackColor = System.Drawing.SystemColors.Window;
      this.ClientSize = new System.Drawing.Size(292, 273);
      this.Name = "Form2";
      this.Text = "Новое окно";
      this.MouseDown += new
System.Windows.Forms.MouseEventHandler(this.Form2_MouseDown);
      this.MouseUp += new
System.Windows.Forms.MouseEventHandler(this.Form2_MouseUp);
      this.MouseMove += new
System.Windows.Forms.MouseEventHandler(this.Form2_MouseMove);

    }
    #endregion

    private void Form2_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
    {
      Form1 parentForm = (Form1)MdiParent;

      switch(parentForm.currentTool)
      {
        case Tools.LINE:
```

```
          DrawLine(new Point(e.X, e.Y));
          break;
        case Tools.ELLIPSE:
          DrawEllipse(new Point(e.X, e.Y));
          break;
        case Tools.TEXT:
          DrawText(new Pointfe.X, e.Y));
          break;
        case Tools.PEN:
          // устанавливаем флаг для начала рисования карандашом
          drawPen -- true;
          break;
      }

      // запоминаем первую точку для рисования
      PreviousPoint.X = e.X;
      PreviousPoint.Y = e.Y;
    }

    private void Form2_MouseUp(object sender,
System.Windows.Forms.MouseEventArgs e)
    {
      drawPen = false;
    }

    private void Form2_MouseMove(object sender,
System.Windows.Forms.MouseEventArgs e)
    {
      // если курсов еще не отпущен
      if(drawPen)
      {
        // создаем объект Pen
        Pen blackPen = new Pen(Color.Black, 3);

        // получаем текущее положение курсора
        Point point = new Point(e.X, e.Y);

        // создаем объект Graphics
        Graphics g = this.CreateGraphics();

        // рисуем линию
        g.DrawLine(blackPen, PreviousPoint, point);

        // сохраняем текущую позицию курсора
        PreviousPoint = point;
      i
```

```
}

void DrawLine(Point point)
{
  // если один раз уже щелкнули
  if (FirstClick == true)
  {
    // создаем объект Pen
    Pen blackPen = new Pen(Color.Black, 3);

    // создаем объект Graphics
    Graphics g = this.CreateGraphics();

    // рисуем линию
    g.DrawLine(blackPen, PreviousPoint, point);

    FirstClick = false;
  }
  else
  {
    FirstClick = true;
  }
}

void DrawEllipse(Point point)
{
  // если один раз уже щелкнули
  if(FirstClick == true)
  {
    // создаем объект Pen
    Pen blackPen = new Pen(Color.Black, 3);

    // создаем объект Graphics
    Graphics g = this.CreateGraphics();

    // рисуем эллипс
    g.DrawEllipse( blackPen,
      PreviousPoint.X, PreviousPoint.Y,
      point.X - PreviousPoint.X, point.Y - PreviousPoint.Y);

    FirstClick = false;
  }
  else
  {
    FirstClick = true;
  }
```

```
    }

    void DrawText(Point point)
    {
      // создаем объект Graphics
      Graphics g = this.CreateGraphics();

      // создаем объект Font
      Font titleFont = new Font("Lucida Sans Unicode", 15);

      // рисуем текст красным цветом
      g.DrawString("Программирование на C#",
        titleFont, new SolidBrush(Color.Red), point.X, point.Y);
    }
  }
}
```

Научно-техническое издание

Лабор Владимир Владимирович

Си Шарп
Создание приложений для Windows

Ответственный за выпуск *В. И. Волкова*
Художественный редактор *А. А. Шуплецов*

Подписано в печать с готовых диапозитивов 05.06.03.
Формат $70x100^{1}/_{16}$. Бумага офсетная. Печать офсетная.
Усл. печ. л. 30,96. Тираж 3 000 экз. Заказ 1581.

ООО «Харвест». Лицензия ЛВ № 32 от 27.08.2002.
РБ, 220013, Минск, ул. Кульман, д. 1, корп. 3, эт. 4, к. 42.

Республиканское унитарное предприятие
«Минская фабрика цветной печати».
РБ, 220024, Минск, ул. Корженевского, 20.