# Программирование для Microsoft® WINDOWS® на C#

Charles Petzold

# Programming Microsoft® WINDOWS® with C#

Microsoft® Press

Чарльз Петцольд

# Программирование для Microsoft® WINDOWS® на C#

Том 1

Москва 2002

РУССКАЯ РЕДАКЦИЯ

**УДК 004.43**
**ББК 32.973.26-018.2**
**П33**

**Петцольд Ч.**

П33 Программирование для Microsoft Windows на C#. В 2-х томах. Том 1./ Пер. с англ. — М.: Издательско-торговый дом «Русская Редакция», 2002. — 576 с.: ил.

**ISBN 5–7502–0210—0**

Ч. Петцольд, известный автор и один из пионеров Windows-программирования, в этой книге подробно и доходчиво рассказывает о возможностях Windows Forms — библиотеки классов Windows нового поколения для платформы .NET. Вы узнаете, как создавать динамические пользовательские интерфейсы, реализовывать графический вывод, управлять клавиатурой, мышью и таймером на C#. Подробное описание языковых конструкций, сравнение их с другими популярными языками программирования и примеры программ помогут вам быстро освоить этот новый объектно-ориентированный язык. В книге подробно рассмотрена иерархия классов .NET Framework, благодаря чему вы сможете приступить к разработке собственных программ с применением Windows Forms.

Том 1 состоит из 11 глав, 3 приложений и снабжен компакт-диском, содержащим примеры программ.

**УДК 004.43**
**ББК 32.973.26-018**

Подготовлено к изданию по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США.



ISBN 0-7356-1370-2 (англ.)
ISBN 5–7502–0210—0

# Содержание

## I ТОМ

### Общее



### Графика



### Пользовательский интерфейс



### Приложения



## II ТОМ

### Графика



### Пользовательский интерфейс

# Оглавление I тома

# Введение

В этой книге рассказывается о том, как писать программы для Microsoft Windows. Способов написания таких программ много, но здесь для этого используются новый объектно-ориентированный язык C# (произносится «Си-шарп») и современная библиотека классов *Windows Forms*, которая входит в состав Microsoft .NET Framework — платформы, представленной публике летом 2000 г. и реализованной спустя примерно полтора года после этого события.

Microsoft .NET Framework — это обширный набор классов, предоставляющий программистам многое из того, что необходимо для создания приложений для Интернета, Web и Windows. Периодика освещала .NET в основном как средство программирования для Web, а в этой книге .NET обсуждается с *другой* стороны. Windows Forms используют для написания как традиционных автономных приложений для Windows (сейчас их иногда называют *клиентскими*), так и интерфейсной части распределенных приложений.

Библиотека Windows Forms предоставляет практически все, что нужно для создания полноценных приложений для Windows. Правда, есть один существенный пробел: в этой библиотеке начисто отсутствует поддержка мультимедиа. В Windows Forms нет даже функции, которая заставила бы пищать динамик компьютера! Я, было, порывался написать собственные мультимедийные классы, но благоразумно (надеюсь) решил повременить с этим, ожидая появления в следующем выпуске Windows Forms гибкой, мощной и простой в использовании поддержки мультимедиа.

Классы, определенные в .NET Framework, нейтральны по отношению к языку. Microsoft выпустила новые версии C++ и Visual Basic, а также новый язык программирования — C#. Другие производители также приспосабливают создаваемые ими языки программирования для работы с классами .NET. Компиляторы новых языков преобразуют исходный текст в .exe-файл, содержащий код на промежуточном языке. В период выполнения этот код компилируется в машинные коды, подходящие для исполняющего программу микропроцессора. Таким образом, .NET потенциально независима от аппаратной платформы.

Я выбрал для этой книги именно C#, так как C# и .NET в прямом смысле слова созданы друг для друга. Поскольку .NET Framework нейтральна по отношению к языку, по этой книге можно учиться писать приложения Windows Forms и на других языках .NET.

## Эволюция Windows-программирования

Первая версия Windows выпущена Microsoft в 1985 г. С тех пор Windows постоянно улучшалась и обновлялась, но самые грандиозные изменения относятся к 1993 (Windows NT) и 1995 (Windows 95) годам, когда Windows перешла с 16-разрядной архитектуры на 32-разрядную.

В первое время после выхода Windows был только один способ создания приложений для Windows — вызов функций интерфейса прикладного программирования (API) Windows из программ на языке C. Хотя Microsoft Pascal также позволял вызывать функции API Windows, этим языком пользовались редко.

За эти годы к программированию для Windows было приспособлено множество других языков, включая Visual Basic и C++. Как C++, так и C# — это объектно-ориентированные языки, поддерживающие большинство типов, операторов, выражений и конструкций C. Поэтому C++ и C# (а также Java) иногда называют языками программирования *на основе C* или языками *семейства C*.

С выходом .NET число подходов, предлагаемых Microsoft для написания Windows-приложений на языках семейства C, увеличилось до трех:

**Эволюция подходов (ориентированных на продукцию Microsoft) к программированию для Windows на языках семейства C**

| Год выхода | Язык | Интерфейс |
|---|---|---|
| 1985 | C | Интерфейс прикладного программирования (API) Windows |
| 1992 | C++ | Библиотека Microsoft Foundation Class (MFC) |
| 2001 | C# или C++ | Windows Forms (часть .NET Framework) |

Я не собираюсь учить вас тому, какой интерфейс или язык выбирать для создания приложений для Windows, — вы должны это сделать самостоятельно в зависимости от поставленной перед вами задачи и доступных ресурсов.

Если вам нужны дополнительные источники по Windows API, то многие нашли полезной в этом плане мою книгу *Programming Windows* (5-е изд., Microsoft Press, 1998).

Мне никогда не нравилась MFC. Еще до выхода этой библиотеки у меня сложилось отрицательное впечатление об ее конструкции, более того, я считаю, что MFC едва ли можно назвать объектно-ориентированной библиотекой. Но это мое личное мнение, другие программисты с успехом используют MFC, и это до сих пор один из самых популярных подходов к программированию для Windows. Хорошее пособие для изучения MFC — книга Джефа Прозиса (Jeff Prosise) *Programming Windows with MFC* (2-е изд., Microsoft Press, 1999). Для более опытных программистов также могу порекомендовать книгу Джеффри Рихтера (Jeffrey Richter) *Programming Applications for Microsoft Windows* (Microsoft Press, 1999)[1].

С моей точки зрения, библиотека Windows Forms организована намного лучше, чем MFC, и в моем представлении она намного ближе к идеальному объектно-ориентированному интерфейсу для Windows. За 14 месяцев, отданных этой книге, использование Windows Forms стало моим любимым подходом к программированию для Windows.

В терминах программирования как MFC, так и Windows Forms работают, вызывая функции Windows API. С точки зрения архитектуры, можно сказать, что эти интерфейсы расположены наверху Windows API. Это интерфейсы более высокого уровня, предназначенные для облегчения программирования для Windows. Вообще MFC и Windows Forms позволяют решать поставленные задачи с меньшим числом операторов, чем при использовании функций API.

Очевидно, что MFC и Windows Forms не только повышают производительность программиста, но и, подобно любому интерфейсу более высокого уровня, обла-

[1] *Рихтер Дж.* Windows для професионалов. М.: Русская Редакция, 2000. — *Прим. перев.*

дают меньшей гибкостью по сравнению с интерфейсом более низкого уровня. Windows API позволяет делать много такого, что невозможно при использовании классов Windows Forms.

К счастью, приложив немного дополнительных усилий, можно вызвать из программы Windows Forms функцию API Windows. Здесь я применял это средство лишь иногда — сталкиваясь со слишком большими пробелами в функциональности .NET. Обычно, следуя принципу своей философии, я стараюсь не нарушать изоляцию внутренних механизмов Windows, предлагаемую Windows Forms.

## Требования к читателю

Для плодотворной работы с этой книгой надо иметь возможность компилировать и исполнять программы на C#. Для компиляции программ нужен компилятор C#, а для их исполнения — исполняющая среда .NET (или CLR), которая представляет собой набор динамически подключаемых библиотек.

Все это есть в Microsoft Visual C#, современной интегрированной среде разработки. Вместо Visual C# можно приобрести более мощную (и, естественно, более дорогую) среду Microsoft Visual Studio .NET, которая в дополнение к C# позволяет программировать на C++ и Visual Basic.

Если вы сторонник варианта «дешево и сердито», можно скачать бесплатный пакет для разработки программ .NET (.NET Framework SDK). В него входят компилятор C# и исполняющая среда .NET. Для этого откройте страницу *http://msdn.microsoft.com/downloads*, выберите слева ссылку Software Development Kits и найдите в списке .NET Framework (помните, что в любой момент содержимое и адрес этого и других Web-узлов, упомянутых в этой книге, может измениться, а в порой узлы могут вовсе исчезнуть).

Эта книга написана, исходя из предположения, что вы умеете программировать хотя бы на C. Знание C++ или Java полезно, но не обязательно. Поскольку C# — новый язык, в первой главе дается краткое введение в C# и разъясняются необходимые понятия объектно-ориентированного программирования. При изложении материала я часто буду обсуждать различные понятия C# по мере знакомства с ними.

Однако эта книга — не исчерпывающее пособие по C#. Для повышения уровня знаний и мастерства владения языком обращайтесь к другим источникам. Несомненно, их станет еще больше по мере роста популярности языка. Книга Тома Арчера (Tom Archer) «Inside C#»[2] дает сведения не только о программировании на C#, но и о более глубоких процессах, а «Microsoft Visual C# Step by Step» (Microsoft Press, 2001) Джона Шарпа (John Sharp) и Джона Джэггера (Jon Jagger) больше похожа на учебник.

Я иногда буду ссылаться на функции API Windows. Как я уже говорил, в случае затруднений обращайтесь за дополнительным разъяснением к моей книге «Programming Windows».

## Требования к системе

Как сказано в предыдущем разделе, для эффективной работы с этой книгой читатель должен иметь возможность компилировать и исполнять программы на C#, к системе же предъявляются такие требования:

[2] *Арчер Т.* «Основы C#», М.: «Русская Редакция», 2001. — *Прим. перев.*

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

Чтобы программы, написанные на C#, можно было исполнять на других компьютерах, на них должна быть установлена исполняющая среда .NET (также называемая *свободно распространяемым пакетом* .NET Framework). Этот пакет поставляется вместе с .NET Framework SDK, Visual C# и Visual Studio .NET и может быть установлен в вышеупомянутые версии ОС Windows, а также в Windows 98 и Windows Millennium Edition (Me).

Чтобы установить файлы примеров с компакт-диска, прилагаемого к этой книге, на жесткий диск, потребуется около 2,1 Мб свободного места (полностью компилированные примеры занимают более 20 Мб).

## Структура книги

В первом выпуске Windows 1.0 весь API умещался в трех динамически подключаемых библиотеках — KERNEL, USER и GDI. Хотя с тех пор число и объем DLL Windows сильно увеличилось, все же полезно делить функции (или классы) Windows на три категории. Во-первых, это функции ядра. Они реализованы во внутренней части ОС и, как правило, отвечают за многозадачность, управление памятью и операции файлового ввода-вывода. Термином *user* здесь обозначен пользовательский интерфейс. К нему относятся функции для создания окон, работы с меню и диалоговыми окнами, а также элементами управления, такими как кнопки и полосы прокрутки. GDI (Graphics Device Interface) — это интерфейс графических устройств, часть Windows, ответственная за вывод графической информации (включая текст) на экран и принтер.

Книга начинается вводными главами. Темы глав с 5 (где рассказывается о рисовании прямых и кривых) по 24 (о буфере обмена Windows) чередуются: главы с нечетными номерами посвящены программированию графики, а с четными — пользовательскому интерфейсу.

Обычно в подобных книгах уделяют мало внимания таким далеким от Windows темам, как файловый ввод-вывод, вычисления с плавающей точкой и манипулирование строками. Однако новизна .NET Framework и C# диктует необходимость включения руководства по соответствующим классам .NET. Некоторые разделы таковыми и являются, а именно три приложения, посвященные обработке файлов, математическим вычислениям и работе со строками. К этим приложениям можно обращаться в любое время после прочтения 1 главы.

Я попытался упорядочить главы и материалы каждой главы так, чтобы построить изложение каждой следующей темы на основе предыдущей и обойтись минимумом «опережающих ссылок». Эта книга написана так, чтобы ее можно было читать последовательно с начала до конца, почти как полную версию «Противостояния»[3] или «Закат и падение Римской Империи»[4].

Хорошо, когда такую книгу можно не только читать подряд, но и использовать как справочник. С этой целью большинство важных методов, свойств и перечис-

[3] Роман С. Кинга . — *Прим. ред.*

[4] Фундаментальный труд историка Е. Гиббона. — *Прим. ред.*

лений, используемых при программировании в Windows Forms, вынесены в таблицы в тех главах, где они обсуждаются. Но даже в такой большой книге невозможно охватить *все* о Windows Forms, поэтому она не может заменить официальную документацию по классам .NET.

Написание программ Windows Forms требует определенных усилий, поэтому вы найдете здесь множество примеров кода в виде законченных программ, фрагменты которых можно совершенно свободно копировать и вставлять в собственные разработки (здесь они приводятся именно *для этого*). Но не распространяйте код или программы в исходном виде, поскольку для этой цели служит книга.

У компилятора C# есть потрясающая функция, позволяющая записывать комментарии в виде тэгов XML. Однако я не использовал ее, поскольку приведенные здесь программы, как правило, содержат мало комментариев, а основное описание кода приводится в тексте.

Возможно, вы знаете, что Visual C# позволяет интерактивно конструировать внешний вид приложений. Вы можете располагать разные элементы управления (кнопки, полосы прокрутки и т. п.) на поверхности окна программы, а Visual C# генерирует соответствующий код. Подобные методики полезны для быстрой разработки диалоговых окон и приложений, богатых элементами управления, однако здесь я проигнорировал эту возможность Visual C#.

В этой книге мы не дадим Visual C# генерировать код за нас, а будем учиться писать его самостоятельно.

## Прилагаемый компакт-диск

На компакт-диске содержатся все программы-примеры. Можно загружать решения (.sln-файлы) или проекты (.csproj-файлы) в Visual C# и компилировать их.

Честно говоря, я никогда особо не пользовался компакт-дисками, прилагаемыми к книгам. При изучении нового языка программирования я предпочитаю набивать исходный текст самостоятельно, даже если это чужая программа. Мне кажется, что так можно быстрее выучить язык, но это моя привычка.

Если вы потеряете или повредите компакт-диск, не шлите мне писем с просьбой заменить его: по условиям договора с издательством я не могу этого сделать. Microsoft Press является единственным распространителем этой книги и прилагаемого к ней диска. Для замены компакт-диска, получения дополнительных сведений или технической поддержки свяжитесь с Microsoft Press (см. контактную информацию ниже в разделе «Техническая поддержка»).

## Техническая поддержка

Я приложил все усилия, чтобы обеспечить точность сведений, изложенных в книге и записанных на компакт-диске. Поправки к этой книге предоставляются Microsoft Press через World Wide Web по адресу:

*http://www.microsoft.com/mspress/support/*

Чтобы напрямую подключиться к Базе знаний Microsoft Press и найти нужную информацию, откройте страницу:

*http://www.microsoft.com/mspress/support/search.asp*

Пожалуйста, присылайте комментарии, вопросы и предложения, касающиеся этой книги или прилагаемого к ней диска, в Microsoft Press:
по обычной почте

```
Microsoft Press
Attn: Programming Microsoft Windows with C# Editor
One Microsoft Way
Redmond, WA 98052-6399
```

или по электронной почте

```
MSPINPUT@MICROSOFT.COM
```

Пожалуйста, обратите внимание, что по этим адресам не предоставляется техническая поддержка. Информацию о технической поддержке C#, Visual Studio или .NET Framework вы найдете на Web-узеле Microsoft Product Support по адресу:

```
http://support.microsoft.com
```

## Особые благодарности

Писателю обычно приходится работать в одиночку, но, к счастью, вокруг всегда находятся люди, облегчающие его труд.

Я хочу поблагодарить моего агента Клодетт Мур (Claudette Moore) из Литературного агентства Мур за то, что она постоянно поддерживала этот проект «на плаву» и взяла на себя труд разобраться со всеми запутанными юридическими вопросами.

Как всегда, сплошным удовольствием была работа с сотрудниками Microsoft Press, которые в очередной раз спасли меня от конфуза. Если бы не редактор проекта, Сэлли Стикни (Sally Stickney) и технический редактор Джин Росс (Jean Ross), примеры программ в этой книге были бы путаными и полны ошибок. Хотя порой мне кажется, что мои редакторы обладают сверхчеловеческими способностями, это, увы, не так. Любые ошибки или невнятные фразы остались исключительно по моей вине.

Да, хочу еще выразить признательность Иоганну Брамсу за музыкальный аккомпанемент к моей работе и Энтони Троллопу, чьи произведения позволили мне отвлекаться от работы во время вечернего чтения.

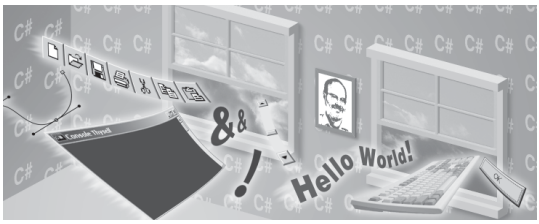
Мои друзья, собирающиеся у меня по воскресеньям, вторникам и четвергам помогали мне и оказывали поддержку, порой незаметную со стороны, но всегда бесценную, и будут делать это дальше.

Больше всех я хотел бы поблагодарить мою невесту Дэйдру, обеспечившую инфраструктуру совсем иного рода (.NET тут ни при чем), позволяющую мне жить, работать и любить.

*Чарльз Петцольд*
*Ноябрь 2001, Нью-Йорк.*

ГЛАВА

# 1




# Работаем с консолью

Учебное пособие Брайена Кернигана и Денниса Ритчи «Язык программирования Си», лаконичное и, наверное, поэтому очень любимое программистами, начинается с примера программы, получившей известность как программа «hello-world»[1]:

```
#include <stdio.h>

main()
{
     printf("hello, world\n");
}
```

Хотя такие программы вряд ли могут задействовать всю мощь современного компьютера, они очень полезны на практике, так как позволяют начинающему программисту убедиться, что компилятор и все относящиеся к нему файлы установлены правильно. Кроме того, программы «hello-world» помогают оценить затраты усилий при работе на данном языке: в одном такая программа может состоять из одной строки, в другом — иметь устрашающие размеры. И, наконец, программы «hello-world» очень полезны авторам книг по программированию, так как показывают, на чем сосредоточиться в начале обучения.

Как знают все С-программисты, точка входа в программу на С — функция с именем *main*, функция *printf* показывает форматированный текст, а файл stdio.h — файл заголовков, содержащий описание *printf* и других стандартных библиотечных функций С. Угловые, простые и фигурные скобки используются, чтобы заключать в них информацию или группировать операторы языка.

---

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd ed. (Englewood Cliffs, NJ: Prentice Hall, 1988). В первой редакции книги эта программа была такой же, но без оператора *#include*.
Имеется перевод на русский: Б. Керниган, Д. Ритчи, *Язык программирования Си* (М.: Финансы и статистика, 1992). — *Прим. перев.*

Традиционные программы «hello-world» создаются для работы в среде, поддерживающей изящный старомодный тип компьютерного интерфейса — *командную строку*, или *консоль*. Этот «чисто текстовый» тип интерфейса восходит к устройству, называемому *телетайпом*, которое в свою очередь, основано на старинном устройстве работы с текстами — *пишущей машинке*. Когда пользователь вводил данные на клавиатуре телетайпа, устройство печатало символы на рулоне бумаги и отправляло их на удаленный компьютер. Удаленный компьютер в ответ отправлял свои символы, которые телетайп принимал и печатал на бумаге. В этой модели ввода/вывода отсутствовала концепция позиционирования текста на странице. Поэтому функция *printf* показывает текст там, где в данный момент находится печатающая головка телетайпа или курсор командной строки видеотерминала.

Интерфейс командной строки существует и в Microsoft Windows в форме окна приложения, называемого приглашением MS-DOS или командной строкой. Хотя с появлением графических интерфейсов интерфейс командной строки считается устаревшим, программы, работающие в режиме командной строки, часто оказываются проще тех, что работают в графической среде. Поэтому с них хорошо начинать изучение новых языков программирования.

## Версия на C#

В этой книге я расскажу о C# (произносится «си-шарп», как тональность «Лунной сонаты» Бетховена[2]). C#, разработанный Андерсом Хейлзбергом (Anders Hejlsberg) из Microsoft, — современный объектно-ориентированный язык программирования, содержащий элементы C, C++, Java, Pascal и даже BASIC. В этой главе дается поверхностное и, конечно же, неполное описание языка.

Файлы исходного кода C# имеют расширение .cs («c sharp»). Моя первая версия программы «hello-world» на C# содержится в файле ConsoleHelloWorld.cs.

**ConsoleHelloWorld.cs**

```
//------------------------------------------------
// ConsoleHelloWorld.cs © 2001 by Charles Petzold
//------------------------------------------------

class ConsoleHelloWorld
{
     public static void Main()
     {
          System.Console.WriteLine("Hello, world!");
     }
}
```

У вас есть пара вариантов компиляции программы в зависимости от того, сколько денег вы готовы потратить и насколько вы требовательны к удобству среды программирования.

---

[2] C# в музыкальной нотации — до-диез; англофоны произносят это как «си-шарп». — *Прим. перев.*

Дешевле всего скачать .NET Framework Software Development Kit (SDK) с *http://msdn.microsoft.com*. При установке SDK устанавливаются и DLL (dynamic-link libraries, динамически подключаемые библиотеки), содержащие исполняющую среду .NET. Здесь же и техническая документация по .NET. Кроме того, вы получите компилятор C#, вызываемый в командной строке, который можно использовать для компиляции приведенных в этой книге программ.

Для написания программ на C# годится любой текстовый редактор, начиная с Microsoft Notepad. Компилятор C# называется csc.exe. ConsoleHelloWorld.cs компилируется такой командной строкой:

```
csc consolehelloworld.cs
```

И все! Этап компоновки отсутствует. (Как вы увидите в главе 2, компиляция программ Windows Forms потребует указания кое-каких дополнительных параметров.) Компилятор создаст файл ConsoleHelloWorld.exe, который можно запускать из командной строки.

Кроме того, можно создавать, компилировать и выполнять программы в Visual C# .NET, новейшей интегрированной среде разработки Microsoft. Visual C# .NET необходима профессиональным разработчикам на C#. Она крайне полезна для определенных типов программ Windows Forms, которые рассматривают окно программы как форму, содержащую управляющие элементы, например, кнопки, поля ввода текста, полосы прокрутки. Но применять Visual C# .NET не обязательно. Я считаю, что одно из самых больших удовольствий программирования для Windows на C# в том, что библиотека Windows Forms не требует использования отдельных файлов. Практически все содержится в файле исходного кода на C#, а все, что нужно для его заполнения, — пораскинуть мозгами и поработать руками.

Далее я опишу действия, которые выполняются для создания программ этой книги в Visual C# .NET. Каждый пример программы из этой книги — это *проект*, под который отводится свой каталог на диске. В Visual C# .NET проекты обычно группируются в *решения*. Например, я создавал решение для каждой из глав этой книги. Решение также является каталогом. Проекты же — подкаталогами каталога решения.

Для создания решения выберите в меню File | New | Blank Solution. В диалоговом окне New Project выберите его местоположение на диске и введите имя решения. Именно так я и создавал решения для каждой из глав этой книги.

После загрузки решения в Visual C# .NET можно создать проекты для этого решения. Выберите File | Add Project | New Project. (Можно также щелкнуть правой кнопкой имя решения в Solution Explorer и выбрать в контекстном меню Add | New Project.) В диалоговом окне Add New Project выберите тип проекта на вкладке Visual C# Projects. Можно выбрать один из нескольких шаблонов. Если вы не хотите, чтобы Visual C# .NET генерировала для вас код (лично я предпочитаю писать собственный), выберите шаблон Empty Project (пустой проект). Именно так я и создавал проекты для этой книги.

При работе с проектом можно выбрать Project | Add New Item для создания нового файла исходного кода на C#. (И здесь тоже можно щелкнуть правой кнопкой имя проекта в Solution Explorer и выбрать этот пункт в контекстном меню.) В диалоговом окне Add New Item в списке Categories выберите Local Project Items.

В разделе Templates выберите Code File. При использовании этого шаблона Visual C# .NET не будет генерировать код для вас.

При любом способе создания и компиляции ConsoleHelloWorld — в командной строке или в Visual C# .NET — файл .exe будет небольшим — 3 или 4 Кб в зависимости от того, помещает ли в него компилятор отладочную информацию. Исполняемый файл состоит из операторов MSIL (Microsoft Intermediate Language). MSIL представлен на рассмотрение ECMA (European Computer Manufacturer's Association, Европейская ассоциация производителей компьютеров) в качестве стандарта под названием CIL (Common Intermediate Language). При запуске программы среда CLR (Common Language Runtime) компилирует промежуточный язык в «родной» машинный код компьютера и подключает соответствующие DLL .NET. Вероятно, сейчас вы используете компьютер архитектуры Intel, поэтому исполняющая среда будет генерировать 32-битный машинный код Intel x86.

Посмотреть на MSIL можно, запустив дизассемблер Intermediate Language Disassembler (ildasm.exe):

```
ildasm consolehelloworld.exe
```

Чтобы получить документацию по набору инструкций MSIL, скачайте файл, описанный сокращением «CIL», с *http://msdn.microsoft.com/net/ecma*. Могут оказаться полезными и другие файлы на этой Web-странице. Можно даже писать код прямо на MSIL и собирать его, используя ассемблер Intermediate Language Assembler (ilasm.exe).

Так как написанные на C# программы компилируются в промежуточный язык, а не напрямую в машинный код, исполняемые файлы независимы от платформы. Возможно, в будущем исполняющую среду .NET перенесут на компьютеры отличной от Intel архитектуры. Если это произойдет, исполняемые файлы, которые вы создаете сегодня, будут на них запускаться. (Позвольте мне добавить «теоретически», чтобы не показаться безнадежно наивным.)

Кроме того, используя .NET Framework и программируя на C#, вы создаете *управляемый код* (managed code) — код, который может исследоваться и анализироваться другими программами для определения его возможностей. Наличие управляемого кода необходимо для обмена двоичными исполняемыми файлами через Интернет.

## Анатомия программы

Давайте еще раз взглянем на программу ConsoleHelloWorld.

Как и в C++ и Java (а также во многих реализациях C), пара символов «/» является началом однострочного комментария. Все, что справа от них, не учитывается при компиляции программы. C# поддерживает и многострочные комментарии, заключаемые в комбинации символов «/*» и «*/». Интересно, что в C# такие комментарии могут содержать операторы XML (Extensible Markup Language), которые в дальнейшем можно отформатировать и задействовать для генерации документации по коду. Это очень удобно, и я советую вам изучить все, что с ней связано, но я вынужден не использовать ее в примерах программ этой книги.

**ConsoleHelloWorld.cs**

```
//--------------------------------------------
// ConsoleHelloWorld.cs © 2001 by Charles Petzold
//--------------------------------------------

class ConsoleHelloWorld
{
     public static void Main()
     {
          System.Console.WriteLine("Hello, world!");
     }
}
```

Точка входа в программу «hello-world» на C# — функция *Main* внутри первой пары фигурных скобок. Подобно C, C++ и Java, C# чувствителен к регистру. Но имя *Main* точки входа в программу на C# пишется с заглавной буквы, тогда как в этих трех языках программирования *main* пишется строчными буквами. Пустые скобки показывают, что *Main* не имеет параметров, а ключевое слово *void* — что она не возвращает значения. Можно также указать, чтобы *Main* принимала массив строк символов в качестве входного параметра и возвращала целое значение. Ключевые слова *public* и *static* я рассмотрю немного позже. Ключевое слово *public* не является здесь строго необходимым; программа будет компилироваться и запускаться и без него.

*Main* располагается внутри определения *класса*. Класс — основной структурный и организационный элемент объектно-ориентированных языков программирования, подобных C#. Проще всего сказать, что класс — совокупность связанных между собой кода и данных. Я назвал этот класс *ConsoleHelloWorld*. В этой книге я буду, как правило, но не всегда определять по одному классу в исходном файле. У файла будет такое же имя, как и у класса, но с расширением .cs. Это соглашение об именах не является необходимым в C#; эта концепция появилась в Java, и мне она нравится. Итак, файл, содержащий класс *ConsoleHelloWorld*, называется ConsoleHelloWorld.cs.

*System.Console.WriteLine* напоминает вызов функции, и это, действительно, так. Данная функция принимает один параметр — текстовую строку — и показывает ее на консоли, в окне командной строки, на вашем допотопном телетайпе или где-то еще. Если вы скомпилируете и запустите программу, она покажет строку:

```
Hello, world!
```

и завершит работу.

Длинное имя функции *System.Console.WriteLine* состоит из таких частей:

- *System* — пространство имен;
- *Console* — класс, определенный в этом пространстве имен;
- *WriteLine* — метод, определенный в этом классе; метод — то же самое, что и функция, процедура или подпрограмма.

# Пространства имен C#

Пространство имен (namespace) — концепция, позаимствованная из C++ и позволяющая обеспечить уникальность всех имен, используемых в конкретной программе или проекте. Иногда программисту при работе над крупным проектом не хватает удобочитаемых глобальных имен или нужны библиотеки классов сторонних разработчиков, в которых конфликтуют имена. Допустим, вы работаете над крупным проектом на C# и приобрели (в виде DLL) пару библиотек классов фирм Bovary Enterprises и Karenina Software. Обе содержат класс *SuperString*, который совершенно по-разному реализован в каждой из DLL, но полезен и в той, и в другой версии. К счастью, такое дублирование — не проблема, так как обе компании следовали принципам C#: использовали пространства имен. Bovary поместила код своего класса *SuperString* в описание имени класса, подобное такому:

```
namespace BovaryEnterprises.VeryUsefulLibrary
{
     class SuperString
     ⋮
}
```

Нечто подобное проделала и Karenina:

```
namespace KareninaSoftware.HandyDandyLibrary
{
     class SuperString
     ⋮
}
```

В обоих случаях сначала идет имя компании, затем название продукта. В ваших программах, использующих эти библиотеки, можно обращаться к конкретному классу *SuperString* через полное имя:

```
BovaryEnterprises.VeryUsefulLibrary.SuperString
```

или:

```
KareninaSoftware.HandyDandyLibrary.SuperString
```

Конечно же, придется много стучать по клавишам, но такое решение определенно работает.

Использование пространств имен было бы довольно вредным явлением, если б не существовал способ уменьшить объем ввода с клавиатуры. Для этого предназначено ключевое слово *using*. Пространство имен указывается один раз в операторе *using*, затем это пространство имен можно не вводить при обращении к его классам. Вот альтернативный вариант программы «hello-world» на C#:

**ConsoleHelloWithUsing.cs**

```
//------------------------------------------------------
// ConsoleHelloWithUsing.cs © 2001 by Charles Petzold
//------------------------------------------------------
using System;
```

*см. след. стр.*

```
class ConsoleHelloWithUsing
{
     public static void Main()
     {
          Console.WriteLine("Hello, world!");
     }
}
```

Если в проекте два различных класса *SuperString*, из затруднения выводит использование псевдонимов с ключевым словом *using*:

```
using Emma = Bovary.VeryUsefulLibrary;
using Anna = Karenina.HandyDandyLibrary;
```

Теперь к этим двум классам можно обращаться как:

```
Emma.SuperString
```

и:

```
Anna.SuperString
```

Подробнее об использовании *using* см. в руководстве по языку C#.

Более 90 пространств имен, определенных в .NET Framework, начинаются со слова *System* и 5 — со слова *Microsoft*. Наиболее важными для этой книги пространствами имен являются само *System*, *System.Drawing*, содержащее различные классы для работы с графикой, и *System.Windows.Forms*.

Пространства имен даже позволяют давать имена, уже задействованные в .NET Framework, собственным классам. Сама .NET Framework также повторно использует некоторые имена классов. Так, она содержит три класса *Timer*, находящиеся в пространствах имен *System.Timers*, *System.Threading* и *System.Windows.Forms*.

Что происходит с классами, определенными без использования пространств имен, такими как *ConsoleHelloWorld* и *ConsoleHelloWithUsing* в наших примерах? Эти имена классов входят в *глобальное пространство имен* (global namespace). Это не проблема для небольших самодостаточных программ, подобных этим. Однако, если я определю в этой книге класс, который может оказаться полезным в программе кого-либо другого, то помещу его в пространство имен *Petzold.ProgrammingWindowsWithCSharp*.

## Консольный ввод-вывод

Пространства имен также играют важную роль в документировании .NET. Например, документацию по классу *Console* нужно искать в документации пространства имен *System*. Вы увидите, что *WriteLine* — не единственный метод вывода класса *Console*. Метод *Write* очень похож на *WriteLine* тем, что также осуществляет вывод на консоль. Отличие в том, что *WriteLine* завершает свой вывод возвратом каретки.

Имеется 18 описаний метода *Write* и 19 — метода *WriteLine*, каждое со своими параметрами. Такое использование нескольких версий одного и того же метода известно как *перегрузка* (overload). Компилятор обычно может определить, какой из перегруженных методов нужно вызвать программе, по количеству и типам передаваемых методу параметров.

Следующая программа демонстрирует три разных способа вывода одной и той же информации.

**ConsoleAdder.cs**

```
//-----------------------------------------
// ConsoleAdder.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;

class ConsoleAdder
{
     public static void Main()
     {
          int a = 1509;
          int b = 744;
          int c = a + b;

          Console.Write("The sum of ");
          Console.Write(a);
          Console.Write(" and ");
          Console.Write(b);
          Console.Write(" equals ");
          Console.WriteLine(c);

          Console.WriteLine("The sum of " + a + " and " + b + " equals " + c);

          Console.WriteLine("The sum of {0} and {1} equals {2}", a, b, c);
     }
}
```

Программа выведет:

```
The sum of 1509 and 744 equals 2253
The sum of 1509 and 744 equals 2253
The sum of 1509 and 744 equals 2253
```

С-программистов обнадежит тот факт, что C# поддерживает знакомый тип *int* и не использует оператор присваивания :=, как Algol или Pascal.

В первом способе программа для показа строки использует вызываемые по отдельности методы *Write* и *WriteLine*, каждый из которых принимает один параметр. *Write* и *WriteLine* могут принимать параметры любых типов и преобразовывать их в строку для показа.

Во втором способе применяется техника, непривычная для программистов на C, зато хорошо знакомая программистам на BASIC, — конкатенация строк при помощи знака «+». C# преобразовывает переменные в строки, объединяет все строки в единое целое и передает результат в функцию *WriteLine*. В третьем способе используется строка форматирования с тремя полями подстановки — {0}, {1} и {2} — для трех других параметров. Поля подстановки могут содержать дополнительную информацию для форматирования. Скажем, {0:C} представляет число как денежную сумму: показывается знак доллара (или другой символ валюты в зависимос-

ти от региональных параметров ОС), запятая, два десятичных знака, если число отрицательное, то строка заключается в скобки. Поле подстановки {0:X8} показывает число в шестнадцатеричном виде, при необходимости дополняя его нулями до 8 знаков. Вот некоторые примеры спецификаций формата в применении к целому числу 12 345:

**Различные спецификации формата в применении к целому числу 12 345**

| Тип форматирования | Код формата | Результат |
|---|---|---|
| Currency (денежные суммы) | C | $12,345.00 |
| | C1 | $12,345.0 |
| | C7 | $12,345.0000000 |
| Decimal (десятичный) | D | 12345 |
| | D1 | 12345 |
| | D7 | 0012345 |
| Exponential (экспоненциальный) | E | 1.234500E+004 |
| | E1 | 1.2E+004 |
| | E7 | 1.2345000E+004 |
| Fixed point (с фиксированной точкой) | F | 12345.00 |
| | F1 | 12345.0 |
| | F7 | 12345.0000000 |
| General (общий) | G | 12345 |
| | G1 | 1E4 |
| | G7 | 12345 |
| Number (числовой) | N | 12,345.00 |
| | N1 | 12,345.0 |
| | N7 | 12,345.0000000 |
| Percent (процент) | P | 1,234,500.00 |
| | P1 | 1,234,500.0 |
| | P7 | 1,234,500.0000000 |
| Hexadecimal (шестнадцатеричный) | X | 3039 |
| | X1 | 3039 |
| | X7 | 0003039 |

Даже если вы не собираетесь использовать вывод на консоль при программировании для .NET, вполне вероятно, что эти спецификации формата пригодятся вам при работе с методом *String.Format*. Точно так же, как *Console.Write* и *Console.WriteLine* являются в .NET аналогами функции C *printf*, метод *String.Format* в .NET аналогичен функции *sprintf*.

# Типы данных C#

Я определил несколько целых чисел с ключевым словом *int* и использовал строки, заключенные в кавычки, так что вы уже знакомы с двумя типами данных, поддерживаемыми C#. На самом деле в C# поддерживаются восемь целочисленных типов данных:

**Целочисленные типы данных C#**

| Количество бит | Со знаком | Без знака |
|---|---|---|
| 8 | *sbyte* | *byte* |
| 16 | *short* | *ushort* |
| 32 | *int* | *uint* |
| 64 | *long* | *ulong* |

Кроме того, в C# поддерживаются два типа данных с плавающей точкой, *float* и *double*, соответствующие стандарту ANSI/IEEE 754-1985 — *IEEE Standard for Binary Floating-Point Arithmetic* (стандарт IEEE двоичной арифметики с плавающей точкой). Количество бит, используемое порядком и мантиссой типов *float* и *double*, таково:

**Количество бит, используемое типами данных C# с плавающей точкой**

| Тип C# | Порядок | Мантисса | Всего бит |
|---|---|---|---|
| *float* | 8 | 24 | 32 |
| *double* | 11 | 53 | 64 |

Кроме того, в C# поддерживается тип *decimal*, использующий 128 бит для хранения данных. В состав числа этого типа входят 96-битная мантисса и десятичный масштабирующий множитель от 0 до 28. Тип данных *decimal* обеспечивает точность около 28 десятичных знаков. Он удобен для хранения и выполнения вычислений над числами с фиксированным количеством десятичных знаков, например, денежных сумм или процентных ставок. Я подробней рассмотрю *decimal*, работу с числами и математические вычисления в C# в приложении Б.

Если вы в тексте программы на C# введете число 3.14, компилятор будет считать, что оно имеет тип *double*. Чтобы указать, что его тип *float* или *decimal*, используйте для типа *float* суффикс *f*, а для *decimal* — суффикс *m*.

Вот небольшая программа, показывающая минимальное и максимальное значения каждого из 11 числовых типов данных.

**MinAndMax.cs**

```
//---------------------------------------
// MinAndMax.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
class MinAndMax
{
     public static void Main()
     {
          Console.WriteLine("sbyte:   {0} to {1}", sbyte.MinValue,
                                                   sbyte.MaxValue);
          Console.WriteLine("byte:    {0} to {1}", byte.MinValue,
                                                   byte.MaxValue);
          Console.WriteLine("short:   {0} to {1}", short.MinValue,
                                                   short.MaxValue);
```

*см. след. стр.*

```
            Console.WriteLine("ushort:  {0} to {1}", ushort.MinValue,
                                                     ushort.MaxValue);
            Console.WriteLine("int:     {0} to {1}", int.MinValue,
                                                     int.MaxValue);
            Console.WriteLine("uint:    {0} to {1}", uint.MinValue,
                                                     uint.MaxValue);
            Console.WriteLine("long:    {0} to {1}", long.MinValue,
                                                     long.MaxValue);
            Console.WriteLine("ulong:   {0} to {1}", ulong.MinValue,
                                                     ulong.MaxValue);
            Console.WriteLine("float:   {0} to {1}", float.MinValue,
                                                     float.MaxValue);
            Console.WriteLine("double:  {0} to {1}", double.MinValue,
                                                     double.MaxValue);
            Console.WriteLine("decimal: {0} to {1}", decimal.MinValue,
                                                     decimal.MaxValue);
        }
    }
```

Как вы заметили, я поставил после каждого типа данных точку и слова *MinValue* или *MaxValue*. Эти два идентификатора являются полями структуры. К концу главы все, что делает эта программа, станет для вас понятным, а пока что просто взглянем на результаты ее выполнения:

```
sbyte:   -128 to 127
byte:    0 to 255
short:   -32768 to 32767
ushort:  0 to 65535
int:     -2147483648 to 2147483647
uint:    0 to 4294967295
long:    -9223372036854775808 to 9223372036854775807
ulong:   0 to 18446744073709551615
float:   -3.402823E+38 to 3.402823E+38
double:  -1.79769313486232E+308 to 1.79769313486232E+308
decimal: -79228162514264337593543950335 to 79228162514264337593543950335
```

В C# поддерживается также тип *bool*, который может принимать только одно из двух значений: *true* или *false*, являющихся ключевыми словами C#. Результаты операций сравнения (==, !=, <, >, <= и >=) — значения типа *bool*. Можно также описать данные типа *bool* явно. Приведение типа *bool* к целым типам допускается (*true* преобразуется в 1, а *false* — в 0), но оно должно быть явным

Тип данных *char* служит для хранения одного символа, а *string* — для хранения строк из нескольких символов. Тип данных *char* отличается от целых типов, и его не следует путать с типами *sbyte* или *byte*. К тому же переменные типа *char* занимают 16 бит (но это не значит, что его можно путать с *short* или *ushort*).

Тип *char* — 16-битный, так как C# использует кодировку Unicode[3] вместо ASCII. Вместо 7-битного представления символов в базовом варианте ASCII и 8 бит на

[3] Дополнительную информацию см. на Web-странице *http://www.unicode.org* или в The Unicode Consortium, The Unicode Standard Version 3.0 (Reading, Mass.: Addison-Wesley, 2000).

символ в расширенных наборах символов, ставших общепринятыми на компьютерах, в Unicode используется целых 16 бит для кодировки одного символа. Это позволяет отобразить все буквы, условные обозначения и другие символы всех письменностей мира, которые могут использоваться при работе с компьютером. Unicode является расширением кодировки ASCII. Его первые 128 символов совпадают с символами ASCII.

Типы данных не обязательно определять в начале метода. Как и в C++, можно определить типы данных в том месте метода, где они потребовались.

Переменную можно определить и сразу же инициализировать:

```
string str = "Hello, World!";
```

Как только переменной типа *string* присвоено значение, ее отдельные символы нельзя изменить, однако можно присвоить ей новое значение. Строки не заканчиваются нулем, а количество символов в переменной типа *string* можно определить так:

```
str.Length
```

*Length* является *свойством* типа данных *string*; концепцию свойств я опишу дальше в этой главе. В приложении B содержится подробная информация по работе со строками в C#.

Чтобы описать переменную-массив, поставьте после типа данных пустые квадратные скобки:

```
float[] arr;
```

Тип данных переменной *arr* — массив чисел с плавающей точкой, но на самом деле *arr* — это указатель. В языке C# массив является *ссылочным* типом (reference type). Это относится и к строке. Другие типы, о которых я рассказывал ранее, являются *размерными* (value types).

Значение переменной *arr* при первоначальном определении — *null*. Для выделения памяти массиву нужно воспользоваться оператором *new* и указать количество элементов в массиве:

```
arr = new float[3];
```

Можно также применить сочетание двух приведенных выше операторов:

```
float[] arr = new float[3];
```

Кроме того, при описании массива можно инициализировать его элементы:

```
float[] arr = new float[3] { 3.14f, 2.17f, 100 };
```

Количество инициализирующих значений должно совпадать с объявленным размером массива. При инициализации массива можно опустить размер:

```
float[] arr = new float[] { 3.14f, 2.17f, 100 };
```

и даже — оператор *new*:

```
float[] arr = { 3.14f, 2.17f, 100 };
```

В дальнейшем в программе можно присвоить переменной *arr* массив типа *float* другого размера:

```
arr = new float[5];
```

При этом выделяется память для хранения пяти значений типа *float*, каждое из этих значений первоначально равно 0.

Вы можете спросить: «А что случилось с первым блоком памяти, выделенным для трех значений типа *float*?» В C# нет оператора *delete*. Поскольку на исходный блок памяти больше не ссылается ни одна из переменных программы, он становится объектом *сборки мусора* (garbage collection). В какой-то момент Common Language Runtime освободит память, изначально выделенную массиву.

Как и в случае строк, количество элементов массива можно определить, используя выражение:

```
arr.Length;
```

Кроме того, C# позволяет создавать многомерные и *невыровненные* (jagged) массивы, являющиеся массивами массивов.

Если не нужно взаимодействие с кодом, написанным на другом языке, указатели в программах на C# требуются редко. По умолчанию параметры передаются методам *по значению*. Это означает, что метод может как угодно изменять параметр, а значение параметра в вызывающем методе не изменится. Чтобы изменить такое поведение параметров, можно использовать ключевые слова *ref* (reference — ссылка) или *out*. Например, определить метод, изменяющий передаваемую в качестве параметра переменную, можно так:

```
void AddFive(ref int i)
{
     i += 5;
}
```

А метод, присваивающий переменной значение, выглядит так:

```
void SetToFive(out int i)
{
     i = 5;
}
```

В первом примере переменной *i* необходимо присвоить значение перед вызовом *AddFive*, тогда метод *AddFive* сможет изменить ее значение. Во втором — *i* при вызове метода может не иметь никакого значения.

Важную роль в C# и .NET Framework играют перечисления. Многие константы .NET Framework определены как перечисления. Приведем пример из пространства имен *System.IO*:

```
public enum FileAccess
{
     Read = 1,
     Write,
     ReadWrite
}
```

Перечисления всегда являются целыми типами данных, по умолчанию они имеют тип *int*. Если не указать значение явно (в данном примере для *Read* оно указано), первому элементу перечисления присваивается нулевое значение. Следующим элементам присваиваются значения в порядке возрастания.

*FileAccess* можно использовать при работе с несколькими классами файлового ввода-вывода. (Файловый ввод-вывод подробно рассмотрен в приложении А). Необходимо указывать имя перечисления, затем, через точку имя элемента:

```
file.Open(FileMode.CreateNew, FileAccess.ReadWrite)
```

*FileMode* — это еще одно перечисление класса *SystemIO*. Если эти два перечисления в методе *Open* поменять местами, компилятор сообщит об ошибке. Перечисления помогают программисту избегать ошибок, связанных с константами.

# Выражения и операторы

Один из самых важных справочников для С-программистов — таблица, в которой приведен порядок выполнения всех операций языка С. (Когда-то можно было купить эту таблицу, отпечатанную на футболке вверх ногами — для удобства чтения.) Аналогичная таблица C# немного отличается от нее в первых двух строках: в них присутствуют некоторые дополнительные операторы и отсутствует оператор «,» (запятая).

Я хотел бы обсудить некоторые тонкости использования двух видов операторов И и ИЛИ, так как они могут вызывать путаницу; по крайней мере, так было у меня при первом знакомстве с ними.

Заметьте, что операторы &, ^ и | называются *логическими* И, исключающим ИЛИ и ИЛИ; в С они назывались *побитовыми*. В C# логические И, исключающее ИЛИ и ИЛИ определены для целых типов данных и для типа *bool*. Для целых типов они выполняются как побитовые, точно так же, как в С. Например, результатом вычисления выражения:

```
0x03 | 0x05
```

будет 0x07. Для переменных или выражений типа *bool* результат выполнения логических операторов также имеет тип *bool*. Результатом логического оператора И является *true*, когда оба операнда — *true*. Результатом логического исключающего ИЛИ является *true*, когда один операнд — *true*, а другой — *false*. Результат логического ИЛИ — *true*, когда хотя бы один из операндов — *true*.

**Порядок вычисления в C#**

| Тип оператора | Оператор | Ассоциативность |
|---|---|---|
| Простой | () [] *f*() . *x*++ *y*++ *new typeof sizeof checked unchecked* | Слева направо |
| Унарный | + ? ! ~ ++*x* ??*x* (*type*) | Слева направо |
| Мультипликативный | * / % | Слева направо |
| Аддитивный | + ? | Слева направо |
| Сдвиг | << >> | Слева направо |
| Отношение | < > <= >= *is as* | Слева направо |
| Равенство | == != | Слева направо |
| Логическое И | & | Слева направо |
| Логическое исключающее ИЛИ | ^ | Слева направо |
| Логическое ИЛИ | \| | Слева направо |
| Условное И | && | Слева направо |

**Порядок вычисления в C#** (*продолжение*)

| Тип оператора | Оператор | Ассоциативность |
|---|---|---|
| Условное ИЛИ | \|\| | Слева направо |
| Условие | ?: | Справа налево |
| Присваивание | = += –= *= /= %= <<= >>= \|= &= ^= | Справа налево |

Операторы && и || называются в языке C *логическими*. В C# они называются *условными* И и ИЛИ и определены только для типа данных *bool*.

C-программисты привыкли использовать && и || в конструкциях вида:

```
if (a != 0 && b >= 5)
```

Кроме того, C-программисты знают, что если результат вычисления первого выражения — *false* (т. е. если *a* равно 0), то второе выражение не вычисляется. Это важно знать, так как второе выражение может выполнять присваивание или вызывать функцию. Аналогично при использовании оператора || второе выражение не вычисляется, если результат первого выражения — *true*.

В C# операторы && и || используются точно так же, как в C. Они называются условными И и ИЛИ, так как второй операнд в них вычисляется только при необходимости.

В C# можно использовать операторы & и | так же, как и && и ||, например:

```
if (a != 0 & b >= 5)
```

При таком использовании операторов & и | в C# вычисляются оба выражения независимо от результата первого выражения.

Второй вариант оператора *if* допустим и в C, и он будет работать так же, как в C#. Однако большинство C-программистов, вероятно, написало бы такой оператор только по ошибке. Этот оператор просто выглядит для меня *неправильным* и заставляет звенеть звоночек в моей голове, так как я приучил себя рассматривать & как побитовое И, а && как логическое И. Но в C результатом отношений или логических выражений является значение типа *int*: 1, если выражение истинно, 0 — в противном случае. Поэтому побитовый оператор И также будет работать правильно.

Программист на C может сделать исходный оператор, использующий &&, немного короче, записав его:

```
if (a && b >= 5)
```

Это прекрасно работает в C, потому что C рассматривает любые ненулевые выражения как истинные. Однако в C# такая конструкция не допускается, так как оператор && определен только для типа *bool*.

Но при использовании побитового оператора И в сокращенной форме выражения программист на C столкнется с большими проблемами:

```
if (a & b >= 5)
```

Если *b* равно 7, выражение справа получит значение 1. Если *a* равно 1, 3 или любому другому нечетному числу, то результатом побитовой операций будет *true*. Если *a* — 0, 2 или любое четное число, то результатом побитового И будет 0, а резуль-

татом всего выражения — *false*. Скорее всего такие результаты совершенно не то, на что рассчитывал программист. Именно поэтому С-программисты столь болезненно реагируют на побитовые операторы И и ИЛИ в логических выражениях. В C# такие операторы запрещены, так как целые числа и значения типа *bool* не могут совместно использоваться в логических операторах И, исключающее ИЛИ и ИЛИ.

В плане приведения типов C# гораздо строже, чем С. Если нужно преобразовать один тип данных в другой, а ограничения C# не позволяют этого, может пригодиться класс *Convert*. Он определен в пространстве имен *System* и содержит многие методы, среди которых, возможно, есть те, что вам подойдут. Если требуется интерфейс с существующим кодом, можно использовать класс *Marshal*, определенный в пространстве имен *System.RuntimeInteropServices*. Он содержит метод *Copy*, позволяющий передавать данные между массивами C# и областями памяти, на которые ссылаются указатели.

## Условия и циклы

В C# поддерживаются многие условные операторы, операторы цикла и управления выполнением программы, применяемые в С. В этом разделе я рассмотрю операторы, содержащие ключевые слова *if*, *else*, *do*, *while*, *switch*, *case*, *default*, *for*, *foreach*, *in*, *break*, *continue* и *goto*.

Конструкции *if* и *else* выглядят точно так же, как в С:

```
if (a == 5)
{
     ⋮
}
else if (a < 5)
{
     ⋮
}
else
{
     ⋮
}
```

Однако в C# выражения в скобках должны иметь тип *bool*. Такое ограничение позволяет избежать общеизвестного подводного камня языка С — ошибочного использования в проверяемом выражении оператора присваивания вместо оператора сравнения:

```
if (a = 5)
```

Такой оператор вызовет в C# ошибку компиляции, и скажите за это спасибо C#.

Но, конечно, ни один компилятор не обеспечивает полной защиты от рассеянности программиста. В одной из своих первых программ на C# я определил переменную *trigger* типа *bool*, но вместо оператора

```
if (trigger)
```

я, чтобы сделать программу чуть понятнее, хотел ввести:

```
if (trigger == true)
```

Но, к сожалению, я ввел:

```
if (trigger = true)
```

Это совершенно корректный, с точки зрения C#, оператор, но, очевидно, он делает не то, что я хотел.

Кроме того, C# поддерживает операторы *do* и *while*. Можно проверять условие перед выполнением блока:

```
while (a < 5)
{
     ⋮
}
```

или после:

```
do
{
     ⋮
}
while (a < 5);
```

Здесь выражение также должно иметь тип *bool*. Во втором случае блок выполнится хотя бы раз независимо от значения *a*.

В конструкции *switch/case* языка C# есть ограничение, которого нет в C. В C допускается оператор:

```
switch (a)
{
case 3:
     b = 7;
               // В C# запрещено прохождение через case.
case 4:
     c = 3;
     break;

default:
     b = 2;
     c = 4;
     break;
}
```

Если *a* равно 3, выполнится первый оператор, затем программа «перешагнет» через *case*, в котором *a* равно 4. Может, вы именно так и задумали, но возможно, что вы просто забыли поставить оператор *break*. Чтобы избежать ситуаций такого рода, компилятор C# сообщает об ошибке. C# позволяет проходить через *case*, только если *case* не содержит операторов. А вот такая конструкция допустима в C#:

```
switch (a)
{
case 3:
case 4:
     b = 7;
     c = 3;
```

```
     break;

default:
     b = 2;
     c = 4;
     break;
}
```

Если требуется нечто более сложное, можно использовать оператор *goto*, описанный в этом разделе далее.

Одна из приятных возможностей C#: в операторе *switch* можно использовать строковую переменную, сравниваемую в операторах *case* со строковыми константами:

```
switch (strCity)
{
case "Boston":
     ⋮
     break;

case "New York":
     ⋮
     break;

case "San Francisco":
     ⋮
     break;

default:
     ⋮
     break;
}
```

Конечно, такой фрагмент кода вызовет ужас у заботящихся о производительности программистов на C или C++. Все эти сравнения строк просто *не могут* быть особо эффективными. На самом деле, благодаря технологии *интернирования строк* (string interning), заключающейся в поддержке таблицы всех уникальных строк, используемых программой, он работает гораздо быстрее, чем может показаться.

Цикл *for* выглядит в C# точно так же, как в C и C++:

```
for (i = 0; i < 100; i += 3)
{
     ⋮
}
```

Как и в C++, в C# широко используется определение параметра цикла прямо в операторе *for*:

```
for (float f = 0; f < 10.05f; f += 0.1f)
{
     ⋮
}
```

Удобным дополнением к этому типу цикла является оператор *foreach*, заимствованный C# из Visual Basic. Предположим, *arr* — массив значений типа *float*. Если требуется показать все его элементы в одной строке, разделив их проблемами, обычно делают так:

```
for (int i = 0; i < arr.Length; i++)
     Console.Write("{0} ", arr[i]);
```

Оператор *foreach*, в котором используется ключевое слово *in*, упрощает эту операцию:

```
foreach (float f in arr)
     Console.Write("{0} ", f);
```

Для параметра цикла *foreach* (в данном случае переменной *f*) надо в операторе *foreach* указать тип данных; в следующем за *foreach* блоке операторов параметр цикла доступен только для чтения. Следовательно, *foreach* не годится для инициализации элементов массива:

```
int[] arr = new int[100];

foreach (int i in arr)          // Так нельзя!
     i = 55;
```

Интересно, что оператор *foreach* может работать не только с массивами, но и с любыми классами, реализующими интерфейс *IEnumerable*, определенный в пространстве имен *System.Collections*. Более ста классов .NET Framework реализуют интерфейс *IEnumerable*. (Здесь я лишь коснусь интерфейсов, а в главе 8 рассмотрю их подробно.)

Оператор *break* обычно используется в конструкциях *switch/case*, а также для выхода из циклов *while, do, for* и *foreach*. Оператор *continue* служит для перехода к концу блока, выполняемого в цикле *while, do, for* или *foreach* и выполнении следующей итерации цикла (если эта итерация должна выполняться согласно условию цикла).

И, наконец, оператор *goto*:

```
goto MyLabel;
     ⋮
MyLabel:
```

полезен для выхода из блоков с большой вложенностью и написания запутанного кода. Кроме того, в C# поддерживается оператор *goto* в конструкции *switch/case* для перехода к другой ветви:

```
switch (a)
{
case 1:
     b = 2;
     goto case 3;

case 2:
     c = 7;
     goto default;
```

```
case 3:
     c = 5;
     break;

default:
     b = 2;
     break;
}
```

Если вместо *break* используется *goto,* то *break* в конце ветви не нужен. Эта возможность компенсирует запрет проходить через *case*.

# Переходим к объектам

В большинстве традиционных процедурных языков, таких как Pascal, Fortran, BASIC, PL/I, C и COBOL, мир делится на код и данные. В основном на них пишут код для «перемалывания» данных.

На всем протяжении истории программирования программисты стремились упорядочить код и данные, особенно в больших программах. Например, взаимосвязанные функции группировались в одном исходном файле. В нем могли находиться и переменные, используемые только этими функциями и нигде более. И, конечно, общепринятым в традиционных языках средством консолидации взаимосвязанных *данных* являются структуры.

Допустим, вы разрабатываете приложение, которое будет работать с датами, в частности, вычислять день года — порядковый номер дня в году. Скажем, для 2 февраля значение дня года равно 33, а для 31 декабря — 366, если год високосный, и 365, если нет. Наверно, вы сочтете разумным описать дату как единую сущность. Например, в C данные, относящиеся к дате, можно описать в структуре с тремя полями:

```
struct Date
{
     int year;
     int month;
     int day;
};
```

Затем можно определить переменную типа *Date*:

```
struct Date today;
```

Можно обращаться к отдельным полям, поставив после имени переменной структурного типа точку и указав имя поля:

```
today.year  = 2001;
today.month = 8;
today.day   = 29;
```

Можно, наоборот, работать с данными структуры как с группой, указывая имя переменной (в данном случае *today*). Например, в C можно одним приемом определить структурную переменную и инициализировать ее:

```
struct Date birthdate = { 1953, 2, 2 } ;
```

Для написания функции вычисления дня года будет полезна небольшая функция, определяющая, каким является год — обычным или високосным:

```
int IsLeapYear(int year)
{
     return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}
```

Ее использует функция вычисления дня года *DayOfYear*:

```
int DayOfYear(struct Date date)
{
     static int MonthDays[12] = {   0,  31,  59,  90, 120, 151,
                                  181, 212, 243, 273, 304, 334 };

     return MonthDays[date.month - 1] + date.day +
                           ((date.month > 2) && IsLeapYear(date.year));
}
```

Обратите внимание, что функция обращается к полям входной структуры, указывая через точку имя поля.

Вот как выглядит полностью работоспособная версия структуры *Date* и относящихся к ней функций на языке C:

**CDate.c**

```
//-----------------------------------
// CDate.c © 2001 by Charles Petzold
//-----------------------------------
#include <stdio.h>

struct Date
{
     int year;
     int month;
     int day;
};
int IsLeapYear(int year)
{
     return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}
int DayOfYear(struct Date date)
{
     static int MonthDays[12] = {   0,  31,  59,  90, 120, 151,
                                  181, 212, 243, 273, 304, 334 };

     return MonthDays[date.month - 1] + date.day +
                           ((date.month > 2) && IsLeapYear(date.year));
}
int main(void)
{
     struct Date mydate;
```

*см. след. стр.*

```
     mydate.month = 8;
     mydate.day   = 29;
     mydate.year  = 2001;

     printf("Day of year = %i\n", DayOfYear(mydate));

     return 0;
}
```

Я поместил функцию *main* в конец программы, чтобы не пришлось использовать упреждающее объявление.

Программа на C имеет такой вид, потому что структуры языка C могут содержать только данные. Код и данные существуют отдельно друг от друга. Однако функции *IsLeapYear* и *DayOfMonth* тесно связаны со структурой *Date*, поскольку они работают с полями структуры *Date*. Поэтому имеет смысл объединить эти функции с самой структурой *Date*. Перемещение функций внутрь структуры превращает программу на C в программу на C++. Вот эта программа на C++:

**CppDateStruct.cpp**

```
//-----------------------------------------------
// CppDateStruct.cpp © 2001 by Charles Petzold
//-----------------------------------------------
#include <stdio.h>

struct Date
{
     int year;
     int month;
     int day;

     int IsLeapYear()
     {
          return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
     }
     int DayOfYear()
     {
          static int MonthDays[12] = {   0,  31,  59,  90, 120, 151,
                                       181, 212, 243, 273, 304, 334 };

          return MonthDays[month - 1] + day + ((month > 2) && IsLeapYear());
     }
};
int main(void)
{
     Date mydate;

     mydate.month = 8;
     mydate.day   = 29;
     mydate.year  = 2001;
```

*см. след. стр.*

```
     printf("Day of year = %i\n", mydate.DayOfYear());

     return 0;
}
```

Объем кода уменьшился. *IsLeapYear* и *DayOfYear* теперь объявлены без параметров. Они могут напрямую обращаться к полям структуры, так как являются частью той же структуры. Теперь эти функции имеют право называться *методами*.

Кроме того, в описании переменной, *mydate* функции *main* исчезло ключевое слово *struct*. Теперь тип *Date* выглядит, как обычный тип данных, а *mydate* — как переменная этого типа. На жаргоне объектно-ориентированного программирования *mydate* называется *объектом* типа *Date* или *экземпляром Date*. Иногда говорят (в основном любители громких слов), что *создается экземпляр* (instantiated) объекта *Date*.

И, наконец, самое важное. Метод *DayOfYear* вызывается аналогично обращению к полям данных структуры. После имени объекта через точку указывается имя метода. Более тонкое изменение заключается в смещении акцента. Раньше мы обращались к функции *DayOfYear*, чтобы она обработала данные, представленные структурой *Date*. Теперь мы обращаемся к структуре *Date*, содержащей реальную календарную дату, чтобы она вычислила свой день года — *DayOfYear*.

Сейчас мы с вами занимаемся объектно-ориентированным программированием, по крайней мере одним из его аспектов. Мы объединяем код и данные в единое целое.

Однако в самых передовых объектно-ориентированных языках элемент программы, содержащий код и данные, называется не *структурой* (*struct*), а *классом* (ключевое слово *class*). Для перехода от *struct* к *class* в C++ требуется добавить всего одну строку кода, ключевое слово *public* в начале описания новоиспеченного класса *Date*.

**CppDateClass.cpp**

```
//-----------------------------------------------
// CppDateClass.cpp © 2001 by Charles Petzold
//-----------------------------------------------
#include <stdio.h>

class Date
{
public:
     int year;
     int month;
     int day;

     int IsLeapYear()
     {
          return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
     }
     int DayOfYear()
```

*см. след. стр.*

```
     {
          static int MonthDays[12] = {   0,  31,  59,  90, 120, 151,
                                       181, 212, 243, 273, 304, 334 };

          return MonthDays[month - 1] + day + ((month > 2) && IsLeapYear());
     }
};
int main(void)
{
     Date mydate;

     mydate.month = 8;
     mydate.day   = 29;
     mydate.year  = 2001;

     printf("Day of year = %i\n", mydate.DayOfYear());

     return 0;
}
```

В C++ и C# классы очень похожи на структуры. И в том, и в другом языке *class* — это *не в точности* то же самое, что и *struct*, но отличия *class* от *struct* в этих двух языках разные. Я рассмотрю отличия *class* и *struct* в C# в этой главе и подробнее — в главе 3. В C++ все поля и методы *struct* по умолчанию открытые (public), т. е. к ним можно обращаться из-за пределов структуры. Например, чтобы я мог обратиться к полям и методам *Date* из функции *main*, они должны быть открытыми. В классах языка C++ все поля и методы по умолчанию закрытые (private), и, чтобы сделать их открытыми, нужно использовать ключевое слово *public*.

Этот пример я выполнял в C++, а не в C#, так как C++ разрабатывался как язык, совместимый с C и позволяющий плавно перейти из мира C в мир C++. Теперь пора выполнить этот пример на C#.

## Программирование в тональности до-диез

На самом деле версия этой программы на C# не так уж сильно отличается от версии на C++.

**CsDateClass.cs**

```
//-----------------------------------------
// CsDateClass.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;

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

*см. след. стр.*

```
            Date mydate = new Date();

            mydate.month = 8;
            mydate.day   = 29;
            mydate.year  = 2001;

            Console.WriteLine("Day of year = {0}", mydate.DayOfYear());
        }
    }
    class Date
    {
        public int year;
        public int month;
        public int day;

        public static bool IsLeapYear(int year)
        {
            return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
        }

        public int DayOfYear()
        {
            int[] MonthDays = new int[] {   0,  31,  59,  90, 120, 151,
                                          181, 212, 243, 273, 304, 334 };

            return MonthDays[month - 1] + day +
                               (month > 2 && IsLeapYear(year) ? 1 : 0);
        }
    }
```

Одно из внесенных мною изменений в том, что метод *Main*, содержащийся теперь в собственном классе, я поместил в начало программы, а класс *Date* — в конец программы. Я был вправе это сделать, так как в C# не требуется упреждающее объявление.

В программе на C++ я определял объект *Date* так:

```
Date mydate;
```

В C# нужна конструкция:

```
Date mydate = new Date();
```

Как и при определении массива, ключевое слово *new* служит для выделения памяти новому объекту типа *Date*. (О скобках после *Date* я расскажу ниже.)

Еще одно изменение в том, что в программе на C# ключевое слово *public* требуется перед каждым полем и методом, к которому обращаются вне класса. Ключевое слово *public* называется *модификатором доступа* (access modifier), так как показывает, как можно обращаться к полям и методам. Альтернативами *public* являются *private* и *protected*, которые я рассмотрю дальше в этой главе.

Заметьте, что метод *IsLeapYear* возвращает значение типа *bool*. В методе *DayOfYear* я использовал условный оператор (?:), чтобы получить значение 1, прибавляемое ко дню високосного года. Я мог бы также привести выражение *bool* к типу *int*.

Давайте немного поговорим на жаргоне. *Date* — это *класс* (class). Класс *Date* содержит пять *членов* (member). Три члена — *year*, *month* и *day* — содержат данные и называются *полями* (fields). Остальные два члена содержат код и называются *методами* (methods). Переменная *mydate* является *объектом* (object) типа *Date*. Кроме того, ее называют *экземпляром* (instance) класса *Date*.

## Статические методы

Переписывая программу с C++ на C#, я внес в нее еще одно изменение: добавил модификатор *static* в определение *IsLeapYear* и добавил в этот метод параметр *year*. Это изменение не необходимо: удаление у метода *IsLeapYear* ключевого слова *static* и параметра не повлияет на работу программы.

Но модификатор *static* имеет настолько большое значение в C# и .NET, что я не хотел откладывать его рассмотрение ни на секунду.

На всем протяжении этой главы я отображал текст на консоли, вызывая метод *WriteLine* класса *Console*:

```
Console.WriteLine(...);
```

Однако при вызове метода *DayOfYear* указывается не класс (в данном случае — *Date*), а *mydate* — объект типа *Date*:

```
mydate.DayOfYear();
```

Уловили разницу? В первом случае указан класс *Console*; во втором случае — объект *mydate*.

Разница как раз и заключается в модификаторе *static*. Метод *WriteLine* в классе *Console* определен как *static*:

```
public static void WriteLine(string value)
```

Статические методы относятся к самому классу, а не к его объектам. Для вызова метода, определенного как *static*, необходимо указать перед ним имя класса. Для вызова метода, не определенного как *static*, нужно указать перед ним имя объекта — экземпляра класса, в котором определен этот метод.

Это различие относится и к членам класса, содержащим данные. Любые поля, определенные как *static*, имеют одно и то же значение для всех экземпляров класса. Вне определения класса к членам, содержащим данные, необходимо обращаться, указывая имя класса, а не имя объекта этого класса. Поля *MinValue* и *MaxValue*, которые я ранее использовал в программе MinAndMax, — статические.

К чему приводит определение *IsLeapYear* как *static*? Во-первых, нельзя вызывать *IsLeapYear*, указывая перед ним экземпляр класса *Date*:

```
mydate.IsLeapYear(1997)     // Не будет работать!
```

Необходимо вызывать *IsLeapYear*, указывая имя класса:

```
Date.IsLeapYear(1997)
```

Внутри определения класса, например в методе *DayOfYear*, перед *IsLeapYear* вообще не нужно ничего ставить. Еще одно следствие: *IsLeapYear* должен иметь параметр — проверяемый год. Преимущество определения *IsLeapYear* как *static* в

том, что для его использования не надо создавать экземпляр *Date*. Аналогично для статических методов, определенных в классе *Console*, не нужно создавать экземпляр класса *Console*. На самом деле вы не сможете создать экземпляр *Console*, но даже если бы это удалось, то все равно было бы бесполезно, так как в классе *Console* нет нестатических методов.

Статический метод не может обращаться к нестатическим методам или полям своего класса. Дело в том, что нестатические поля различны для разных экземпляров класса, а нестатические методы возвращают различные значения для разных экземпляров класса. Когда вы ищете что-нибудь в справочнике по .NET Framework, всегда обращайте внимание, определено ли оно как *static* или нет. Это крайне важно. Я также постараюсь быть в этой книге очень аккуратным и обращать внимание, когда что-либо определено как *static*.

Поля также могут определяться как *static*, и тогда они совместно используются всеми экземплярами класса. Статические поля хорошо подходят для массивов, которые требуется инициализировать постоянными значениями, как в случае массива *MonthDays* в программе CsDateClass. Как видно из этой программы, массив повторно инициализируется при каждом вызове метода *DayOfYear*.

# Обработка исключений

Разные ОС, графические среды, библиотеки и функции по-разному сообщают об ошибках. Одни возвращают логические значения, другие — коды ошибок, третьи — значение NULL, четвертые подают звуковой сигнал, пятые приводят к аварийному сбою системы.

В C# и .NET Framework для сообщения об ошибках предпринята попытка унифицировать использование технологии *структурной обработки исключений* (structured exception handling).

Чтобы изучить ее, начнем с того, что в программе CsDateClass присвоим полю *month* объекта *Date* значение 13:

```
mydate.month = 13;
```

Перекомпилируйте и запустите программу. Если откроется диалоговое окно выбора отладчика, щелкните No. Вы получите в командной строке сообщение:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the
bounds of the array.
   at Date.DayOfYear()
   at CsDateClass.Main()
```

При компиляции с установкой режима отладки информация будет конкретнее: вы увидите номера строк исходного кода. Но в любом случае выполнение программы завершится преждевременно.

Заметьте, что сообщение правильно: индекс массива *MonthDays*, действительно, вне границ массива. В языке C проблемы такого рода приводили к другим видам ошибок, например, к переполнению стека. C# выполняет проверку корректности индекса перед доступом к элементу массива по его индексу. Программа реагирует на неправильный индекс, инициируя простой процесс, который называется *выбросом* (throwing) или *генерацией* исключения.

В C# есть очень полезная возможность — сделать так, чтобы программы сами определяли возникновение исключений и обрабатывали их. Когда программа проверяет возникновение исключения, говорят, что она *перехватывает* (catch) исключение. Для перехвата исключения операторы, которые могут вызвать исключение, помещаются в блок *try*, а операторы, выполняемые при обработке исключения, — в блок *catch*. Например, чтобы учитывать возможность указания неправильной даты, в программу CsDateClass можно поместить такой код:

```
try
{
     Console.WriteLine("Day of year = {0}", mydate.DayOfYear());
}
catch (Exception exc)
{
     Console.WriteLine(exc);
}
```

Класс *Exception* определен в пространстве имен *System*, а *exc* — объект типа *Exception*, определенный в программе. Этот объект получает информацию об исключении. В данном примере я просто передал *exc* методу *Console.WriteLine* в качестве параметра. При задании неправильного месяца этот оператор покажет точно такой же блок текста с описанием ошибки, как тот, что я показывал ранее. Но теперь программа не завершается преждевременно, и можно обрабатывать ошибку более изящно.

Даже одна строка кода может вызвать исключения разных типов. Поэтому можно определить несколько разных блоков *catch*:

```
try
{
     ⋮
}
catch (NullReferenceException exc)
{
     ⋮
}
catch (ArgumentOutOfRangeException exc)
{
     ⋮
}
catch (Exception exc)
{
     ⋮
}
```

Обратите внимание, что исключение самого общего типа находится в конце.

Можно также добавить блок *finally*:

```
try
{
     ⋮
}
```

```
catch (Exception exc)
{
     ⋮
}
finally
{
     ⋮
}
```

Независимо от того, возникнет исключение или нет, код блока *finally* все равно выполнится: после кода блока *catch* (если исключение возникнет) или после кода блока *try* (если нет). В блок *finally* можно поместить, например, код освобождения ресурсов.

Вы можете спросить: «А зачем блок *finally*? Разве нельзя просто поместить код освобождения ресурсов после блока *catch*?» Можно. Но вы можете закончить блок *try* или *catch* оператором *goto*. В этом случае код блока *finally* все равно выполнится перед выполнением оператора *goto*.

Можно также пропустить блок *catch*:

```
try
{
     ⋮
}
finally
{
     ⋮
}
```

В этом случае вы увидите окно с информацией об отладчике и текстовым описанием исключения (тем же, что мы показывали при помощи *Console.WriteLine*), затем выполнится код блока *finally*, и программа завершится нормально.

# Выброс исключений

В программе работы с датами CsDateClass меня беспокоит то, что мы не добрались до истинной причины проблем. Метод *DayOfYear* выбрасывает исключение из-за того, что индекс массива *MonthDays* находится вне границ. Но настоящая проблема возникает в программе раньше и связана с оператором, который я предложил добавить в программу:

```
mydate.month = 13;
```

Сразу после его выполнения мы имеем дело с объектом *Date*, содержащим неправильную дату. В этом и заключается настоящая проблема. Просто получилось так, что *DayOfYear* оказался первым методом, который отрицательно реагирует на эту проблему. Но допустим, что вы введете в программу другой оператор:

```
mydate.day = 47;
```

Метод *DayOfYear* выполнится без сообщений об ошибках и посчитает результат, несмотря на некорректность даты.

Есть ли способ встроить в класс защиту от присваивания полям некорректных значений программой, использующей этот класс? Проще всего описать поля как *private*, а не *public*:

```
private int year;
private int month;
private int day;
```

Модификатор *private* делает эти три поля доступными только методам, находящимся внутри определения класса *Date*. На самом деле в C# атрибут *private* используется по умолчанию, так что для внесения этих изменений достаточно просто удалить атрибут *public*:

```
int year;
int month;
int day;
```

Конечно, такое изменение порождает уже другую проблему. Как программа, использующая класс *Date*, должна устанавливать значения года, месяца и дня?

Одно из решений, которые могут прийти в голову, — определить методы класса *Date*, предназначенные специально для присваивания значений этих трех полей и для получения ранее присвоенных значений полей. Вот, например, два простых метода для присваивания и получения значения закрытого поля *month*:

```
public void SetMonth (int month)
{
     this.month = month;
}
public int GetMonth ()
{
     return month;
}
```

Заметьте: оба метода определены как *public*. Кроме того, я дал параметру метода *SetMonth* имя, совпадающее с именем поля! При этом перед именем поля нужно указывать слово *this* и точку. Внутри класса ключевое слово *this* означает экземпляр класса, вызывающего метод. В статических методах ключевое слово *this* не допускается.

Вот версия *SetMonth*, проверяющая правильность значения месяца:

```
public void SetMonth (int month)
{
     if (month >= 1 && month <= 12)
          this.month = month;
     else
          throw new ArgumentOutOfRangeException("Month");
}
```

В этом примере показан синтаксис выброса исключения. Я выбрал исключение *ArgumentOutOfRangeException*, так как оно наиболее адекватно идентифицирует проблему. Ключевое слово *new* используется для создания нового объекта типа *ArgumentOutOfRangeException*. Этот объект — то, что блок *catch* получает в ка-

честве параметра. Параметром *ArgumentOutOfRangeException* является текстовая строка, идентифицирующая вызвавший проблему параметр. При выводе сообщения об исключении эта строка показывается вместе с другой информацией об ошибке.

В C# имеется достойная альтернатива методам *Get* и *Set*. Когда вы собираетесь писать методы, начинающиеся со слов *Get* и *Set*, более того, когда вы собираетесь писать любой метод, возвращающий информацию об объекте и не требующий параметров, вам следует подумать о такой возможности C#, как *свойство* (property).

## Получение и присваивание значений свойств

Как вы уже знаете, в классах C# могут присутствовать члены, содержащие данные, называемые *полями*, и члены, содержащие код, — *методы*. Кроме того, в классах C# может быть еще один вид членов, содержащих код. Эти члены называются *свойствами* и играют крайне важную роль в .NET Framework.

Свойства как бы размывают различия между кодом и данными. Для использующих класс программ свойства выглядят как поля с данными и часто именно так и рассматриваются. Однако внутри класса свойства, конечно же, являются кодом. Во многих программах открытое свойство обеспечивает другим классам доступ к закрытому полю класса. Свойство имеет преимущество перед полем, так как в него можно поместить проверку допустимости значения.

Некоторые программисты на C# (в том числе и я) дают закрытым полям имена, начинающиеся со строчных букв, а открытым свойствам — имена, начинающиеся с заглавных букв. Ниже приведено простейшее определение свойства *Month*, обеспечивающего доступ к закрытому полю *month*:

```
public int Month
{
     set
     {
          month = value;
     }
     get
     {
          return month;
     }
}
```

Программа, использующая класс с таким свойством, обращается к свойству точно так же, как если бы это было поле:

```
mydate.Month = 7;
```

или:

```
Console.WriteLine(mydate.Month);
```

или:

```
mydate.Month += 2;
```

В последнем примере значение *Month* увеличивается на 2. Смотрите, насколько синтаксис этого оператора прозрачнее, чем синтаксис аналогичного оператора, использующего методы *SetMonth* и *GetMonth*, с которыми мы работали ранее:

```
mydate.SetMonth(mydate.GetMonth() + 2);       // Хорошо, что мы от этого
                                               // избавились!
```

Рассмотрим определение этого свойства подробно. Ключевое слово *public* означает, что это свойство доступно вне класса. Тип данных *int* — что свойство является 32-битным целым. Само свойство имеет имя *Month*.

Внутри тела свойства содержатся два *аксессора* (accessors) — *set* и *get*. Не обязательно использовать оба. У многих свойств есть только открытый *get*-аксессор, а *set* или вообще не определен, или определен как *private*. Такие свойства называются *неизменяемыми* (read-only). Можно также создать свойство с *set*-аксессором и без *get*-аксессора, но это требуется гораздо реже.

Внутри определения *set*-аксессора специальное слово *value* служит для указания значения, присваиваемого свойству оператором:

```
mydate.Month = 7;
```

Аксессор *get* всегда должен содержать оператор *return*, чтобы вернуть значение программе, использующей свойство.

Следующая программа использует свойства *Year*, *Month* и *Day* и выполняет в *set*-аксессорах проверку допустимости значений.

**CsDateProperties.cs**

```
//---------------------------------------------
// CsDateProperties.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;

class CsDateProperties
{
     public static void Main()
     {
          Date mydate = new Date();

          try
          {
               mydate.Month = 8;
               mydate.Day   = 29;
               mydate.Year  = 2001;

               Console.WriteLine("Day of year = {0}", mydate.DayOfYear);
          }
          catch (Exception exc)
          {
               Console.WriteLine(exc);
          }
     }
```

*см. след. стр.*

```
}
class Date
{
                                                                    // Поля
     int year;
     int month;
     int day;
     static int[] MonthDays = new int[] {   0,  31,  59,  90, 120, 151,
                                          181, 212, 243, 273, 304, 334 };

                                                                    // Свойства
     public int Year
     {
          set
          {
               if (value < 1600)
                    throw new ArgumentOutOfRangeException("Year");
               else
                    year = value;
          }
          get
          {
               return year;
          }
     }
     public int Month
     {
          set
          {
               if (value < 1 || value > 12)
                    throw new ArgumentOutOfRangeException("Month");
               else
                    month = value;
          }
          get
          {
               return month;
          }
     }
     public int Day
     {
          set
          {
               if (value < 1 || value > 31)
                    throw new ArgumentOutOfRangeException("Day");
               else
                    day = value;
          }
          get
```

*см. след. стр.*

```
            {
                return day;
            }
        }
        public int DayOfYear
        {
            get
            {
                return MonthDays[month - 1] + day +
                                (month > 2 && IsLeapYear(year) ? 1 : 0);
            }
        }
                                                                // Метод
        public static bool IsLeapYear(int year)
        {
            return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
        }
    }
```

Я оставил код в блоках *try* и *catch*, так что вы можете поэкспериментировать с некорректными датами. Кстати, для свойства *Year* я установил минимальное значение 1600, так как для более ранних дат метод *IsLeapYear* не имеет особого смысла. Осталась нерешенной одна проблема: метод не проверяет согласованность полей между собой. Например, можно задать дату 31 февраля. Такая проверка согласованности наложила бы ограничения на порядок присвоения свойств, так что я обойдусь без нее.

Кроме того, я преобразовал *DayOfYear* из метода в неизменяемое свойство, потому что значение дня года больше похоже на свойство даты, чем на метод. Иногда бывает сложно определить, что использовать — метод или свойство? Единственное очевидное правило: есть параметр — используй метод.

# Конструкторы

Снова вернемся к версии программы на C. Я уже упоминал, что при определении структурной переменной можно инициализировать поля структуры:

```
struct Date birthdate = { 1953, 2, 2 } ;
```

Однако я не рассматривал, можно ли применять этот способ в разных версиях C. Структуры или классы языка C++ можно инициализировать таким образом, но тогда инициализация будет зависеть от количества полей в *struct* или *class* и порядка, в котором расположены эти поля, так что это не очень хорошая идея. В C# такая инициализация не допускается. Но было бы очень хорошо, если бы в C# было *нечто*, позволяющее решать подобные задачи.

Еще один момент. В предыдущей версии программы на C# мы реализовали проверку допустимости значений во всех *set*-аксессорах свойств класса. Однако ситуация, когда класс содержит некорректную дату, по-прежнему возможна — это ситуация, когда объект только что создан:

```
Date mydate = new Date();
```

Можно решить эти две проблемы при помощи средства, называемого *конструктором* (constructor). Конструктор — метод класса, выполняемый при создании объекта этого класса. Как видите, после слова *new* в конструкции:

```
Date mydate = new Date();
```

идет нечто, похожее на вызов метода без параметров. И это так! Это вызов конструктора класса *Date* по умолчанию. У каждого класса есть конструктор по умолчанию, существующий независимо от того, определен он явно или нет. Но если явно определить конструктор класса *Date* по умолчанию, то можно добиться, что объект *Date* всегда будет содержать допустимую дату.

Можно также определить конструкторы с одним или несколькими параметрами. Например, в классе *Date* можно определить конструктор с тремя параметрами, который будет инициализировать объект *Date* конкретной датой. Этот конструктор позволит создавать объект *Date* таким образом:

```
Date birthdate = new Date(1953, 2, 2);
```

Внутри класса конструктор выглядит как метод за исключением того, что имеет то же имя, что и класс, в котором определен, и того, что в конструкторе не определяется тип возвращаемого значения. Если указать в конструкторе тип возвращаемого значения или определить другой метод, не указав тип возвращаемого значения, компилятор сообщит об ошибке. Это правильно: ведь если при определении конструктора имя класса указано неверно, вы об этом узнаете при компиляции.

Вот простой пример конструктора, параметры которого определяют дату:

```
public Date(int year, int month, int day)
{
     this.year  = year;
     this.month = month;
     this.day   = day;
}
```

Но в нем не используется реализованная в свойствах проверка ошибок. Правильнее присвоить значения свойствам, а не полям:

```
public Date(int year, int month, int day)
{
     Year  = year;
     Month = month;
     Day   = day;
}
```

Но можно пойти еще дальше. Можно выполнить проверку совместимости трех параметров конструктора.

А как быть с конструктором по умолчанию? Принято определять конструкторы классов по умолчанию, присваивающие объектам нулевые значения или значения, более-менее эквивалентные нулевым. Для класса *Date* таким значением, вероятно, будет 1 января 1600 года, так как это самая ранняя допустимая дата. Вот новая версия программы:

**CsDateConstructors.cs**

```
//-----------------------------------------------
// CsDateConstructors.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;

class CsDateConstructors
{
     public static void Main()
     {
          try
          {
               Date mydate = new Date(2001, 8, 29);

               Console.WriteLine("Day of year = " + mydate.DayOfYear);
          }
          catch (Exception exc)
          {
               Console.WriteLine(exc);
          }
     }
}
class Date
{
                                                              // Поля
     int year;
     int month;
     int day;
     static int[] MonthDays = new int[] {   0,  31,  59,  90, 120, 151,
                                          181, 212, 243, 273, 304, 334 };

                                                              // Конструкторы
     public Date()
     {
          Year = 1600;
          Month = 1;
          Day = 1;
     }
     public Date(int year, int month, int day)
     {
          if ( (month == 2 &&  IsLeapYear(year) && day > 29) ||
               (month == 2 && !IsLeapYear(year) && day > 28) ||
              ((month == 4 || month == 6 ||
                              month == 9 || month == 11) && day > 30))
          {
               throw new ArgumentOutOfRangeException("Day");
          }
          else
          {
```

*см. след. стр.*

```
            Year  = year;
            Month = month;
            Day   = day;
        }
    }
                                                        // Свойства
    public int Year
    {
        set
        {
            if (value < 1600)
                throw new ArgumentOutOfRangeException("Year");
            else
                year = value;
        }
        get
        {
            return year;
        }
    }
    public int Month
    {
        set
        {
            if (value < 1 || value > 12)
                throw new ArgumentOutOfRangeException("Month");
            else
                month = value;
        }
        get
        {
            return month;
        }
    }
    public int Day
    {
        set
        {
            if (value < 1 || value > 31)
                throw new ArgumentOutOfRangeException("Day");
            else
                day = value;
        }
        get
        {
            return day;
        }
    }
    public int DayOfYear
```

*см. след. стр.*

```
        {
            get
            {
                return MonthDays[month - 1] + day +
                                    (month > 2 && IsLeapYear(year) ? 1 : 0);
            }
        }
                                                                    // Метод
        public static bool IsLeapYear(int year)
        {
            return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
        }
    }
```

## Экземпляры и наследование

Вполне возможно, что по прошествии некоторого времени пользования классом вы подумаете: «Этот класс очень хорош, но лучше, если б...» Если у вас есть исходный код класса, можете просто открыть его в редакторе, добавить новый метод, перекомпилировать и начать использовать. Но у вас может и не быть исходного кода. Возможно, что вам доступна только скомпилированная версия класса в виде DLL.

Может случиться и так: вы хотите, чтобы некоторые функции класса выполнялись иначе. Но класс уже используется в существующем виде другими приложениями и вполне их устраивает. Изменения требуются только для одного нового приложения, и вы предпочли бы не «засорять» код исходной версии.

В объектно-ориентированных языках, подобных C#, реализована возможность *наследования* (inheritance). Можно определить новый класс, основанный на существующем классе. Говорят, что вы *наследуете* (inherit) класс от существующего класса или создаете *подкласс* (subclass) существующего класса. В новом классе должны содержаться только новые данные и код. Все классы C# и .NET Framework являются наследниками класса *Object* или классов-наследников *Object*. Говорят также, что все классы в конечном счете являются *производными* от *Object*.

Давайте создадим новый класс *DatePlus*, унаследованный от *Date*. *DatePlus* будет обладать новым свойством *DaysSince1600*. Благодаря наличию этого свойства, можно сделать, чтобы *DatePlus* вычислял разницу в днях между двумя датами.

Вот программа, в которой определен класс *DatePlus*.

**CsDateInheritance.cs**

```
//-------------------------------------------
// CsDateInheritance.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;

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

*см. след. стр.*

```
        {
            DatePlus birth = new DatePlus(1953, 2, 2);
            DatePlus today = new DatePlus(2001, 8, 29);

            Console.WriteLine("Birthday = {0}", birth);
            Console.WriteLine("Today = " + today);
            Console.WriteLine("Days since birthday = {0}", today - birth);
        }
    }
    class DatePlus: Date
    {
        public DatePlus() {}
        public DatePlus(int year, int month, int day): base(year, month, day) {}

        public int DaysSince1600
        {
            get
            {
                return 365 * (Year - 1600) +
                       (Year - 1597) / 4 -
                       (Year - 1601) / 100 +
                       (Year - 1601) / 400 + DayOfYear;
            }
        }
        public override string ToString()
        {
            string[] str = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                             "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

            return String.Format("{0} {1} {2}", Day, str[Month - 1], Year);
        }
        public static int operator -(DatePlus date1, DatePlus date2)
        {
            return date1.DaysSince1600 - date2.DaysSince1600;
        }
    }
```

Эту программу необходимо компилировать вместе с файлом CsDateConstructors.cs — самой последней реализацией класса *Date*. Так как имеется два класса, содержащих метод *Main*, нужно указать компилятору, какой из классов, содержащих метод *Main*, использовать для входа в программу.

При компиляции в командной строке надо ввести:

```
csc CsDateConstructors.cs CsDateInheritance.cs /main:CsDateInheritance
```

При этом нужно быть внимательным к регистру букв. Имена файлов можно вводить в любом регистре, но параметр */main* обращается к классу, и регистр символов этого параметра должен соответствовать имени класса, определенному в файле. В случае Visual C# .NET нужно добавить CsDateConstructors.cs в проект CsDateInheritance. Для этого выберите в меню Project | Add Existing Item. При выборе файла

CsDateConstructors.cs в диалоговом окне Add Existing Item щелкните стрелку рядом с кнопкой Open и выберите Link File. Этот параметр позволяет избежать создания копии файла CsDateConstructors.cs, а также проблем, возникающих в случае, когда версия одной из копий этого файла изменилась, а другую при этом забыли обновить.

Обратите внимание на первую строку определения класса *DatePlus*:

```
class DatePlus: Date
```

Это означает, что *DatePlus* наследуется от *Date*. Классу *DatePlus* не нужно выполнять в конструкторах специальных действий. Поэтому для него определен конструктор по умолчанию с пустым телом:

```
public DatePlus() {}
```

При создании экземпляра класса вызываются конструкторы по умолчанию всех объектов, от которых унаследован класс, начиная с конструктора по умолчанию класса *Object* и кончая конструктором по умолчанию класса объекта, который вы создаете.

Это правило неприменимо к конструкторам не по умолчанию. Конструктор с тремя параметрами не выполняет никаких специальных действий в *DatePlus*, но нужно определить его в классе и явно вызвать конструктор *базового* (base) класса, от которого наследуется *DatePlus*, — класса *Date*. Определение конструктора не по умолчанию имеет такой синтаксис:

```
public DatePlus(int year, int month, int day): base(year, month, day) {}
```

Так как конструктор не делает в классе *DatePlus* ничего специфического, его тело пусто.

В классе *DatePlus*, кроме свойства *DaysSince1600*, реализованы две изящные возможности. Во-первых, в *DatePlus* определен оператор вычитания (–) для объектов этого класса. Это называется *перегрузкой* (overloading) *оператора*. Обычно оператор минус определяется только для чисел, но здесь он хорошо подходит и для работы с датами. Тело этого перегруженного оператора довольно простое: в нем просто вычитаются *DaysSince1600* двух дат.

Например, если определить два объекта *DatePlus*:

```
DatePlus birth = new DatePlus(1953, 2, 2);
DatePlus today = new DatePlus(2001, 8, 29);
```

то можно посчитать разницу в днях между этими датами при помощи выражения

```
today - birth
```

Заметьте: я не реализовал в этом классе перегруженный оператор сложения. Не имеет смысла складывать друг с другом две даты. Однако, я мог бы реализовать сложение даты и числа, результатом которого является новая дата. Но тогда потребовалось бы писать код, чтобы преобразовать новое значение *DaysSince1600* в дату. Зато довольно легко реализовать операторы сравнения (<, >, <= и >=).

Во-вторых, я уже упоминал, что все объекты в конечном счете наследуются от *Object*. В классе *Object* реализован метод *ToString*, предназначенный для преобразования объекта в читабельную текстовую строку. Мы уже использовали *ToString*.

При конкатенации числовой переменной с текстовой строкой автоматически вызывается метод *ToString* этой переменной. При передаче объекта в метод *Console.WriteLine* также вызывается метод *ToString* объекта.

Но по умолчанию метод *ToString* класса *Object* возвращает имя класса, например строку «DatePlus». Ничего страшного: ведь любой класс, происходящий от *Object* (т. е. любой класс C#) может *переопределить* (override) метод *ToString* класса *Object* своим собственным. В классе *DatePlus* реализован собственный метод *ToString*, использующий статический метод *String.Format* для преобразования даты в текстовую строку. Стало возможным вызвать метод *Console.WriteLine*, указав в качестве параметра объект *DatePlus*, и показать форматированную дату. Вывод программы CsDateInheritance выглядит так:

```
Birthday = 2 Feb 1953
Today = 29 Aug 2001
Days since birthday = 17740
```

Теперь мы готовы более подробно рассмотреть модификаторы доступа. Если поле, свойство или метод определены как *private*, они видимы и доступны только внутри класса. Если поле, свойство или метод определены как *public*, они видимы и доступны в других классах. Если поле, свойство или метод определены как *protected*, они видимы и доступны только внутри класса и любого класса, унаследованного от этого класса.

Метод *ToString* класса *Object* определен с модификатором *virtual*. Метод, определенный как *virtual*, может переопределяться наследниками класса. При переопределении метода используется модификатор *override*, показывающий, что метод заменяется собственной версией, реализованной в этом классе. Модификатор *override* необходим, чтобы нельзя было ошибиться, случайно переопределив виртуальный метод, когда этого не планировалось.

Кроме того, в классах можно переопределять методы, не объявленные как *virtual*. В этом случае новый метод должен содержать модификатор *new*.

Кроме *ToString*, класс *Object* содержит несколько других методов, в том числе *GetType*. *GetType* возвращает объект типа *Type*. *Type* — это класс, определенный в пространстве имен *System*, позволяющий получить информацию об объекте, в частности, о его методах, свойствах и полях. Оператор C# *typeof* также возвращает объект типа *Type*. Отличие между ними в том, что *GetType* применяется к объекту, а *typeof* — к классу. В методе *Main* программы CsDataInheritance результатом выражения:

```
today.GetType() == typeof(DatePlus)
```

будет *true*.

# Общая картина

Документация библиотек классов в .NET Framework упорядочена по пространствам имен. Каждое пространство имен логически объединяет классы и другие элементы и реализуется в конкретной DLL.

В каждом из пространств имен можно встретить элементы пяти видов. Только эти пять видов элементов определяются в C# на внешнем уровне:

- *class* (класс), с которым вы уже познакомились;
- *struct* (структура), во многом похожая на класс;
- *interface* (интерфейс) — аналогичен *class* и *struct*, но в нем определяются только сами методы, а их тела не определяются (пример интерфейса см. в главе 8);
- *enumeration* (перечисление) — список констант с заранее определенными целыми значениями;
- *delegate* (делегат) — прототип вызова метода.

Классы и структуры выглядят в C# очень похожими. Однако *class* является *ссылочным типом*. Это значит, что объект на самом деле служит указателем на выделенный ему блок памяти. А *struct* является *размерным типом*, более похожим на обычную числовую переменную. Подробнее я рассмотрю различия между ними в главе 3. О делегатах я расскажу в главе 2; они обычно используются вместе с *событиями* (event).

Одни классы .NET Framework содержат статические методы и свойства, к которым обращаются, указывая имя класса и имя метода или свойства. Чтобы использовать другие классы, нужно создавать их экземпляры в приложениях Windows Forms. Третьи классы .NET Framework наследуются вашими приложениями.

В состав классов и структур могут входить такие члены:

- *поля* — объекты определенных типов;
- *конструкторы*, выполняемые при создании объекта;
- *свойства* — блоки кода с аксессорами *set* и *get*;
- *методы* — функции, принимающие аргументы и возвращающие значения;
- *операторы*, реализующие стандартные операции, такие как + и –, определяемые для объекта или *приведения типов* (cast);
- *индексаторы* (indexer), позволяющие обращаться к объекту как к массиву;
- *события*, о которых я расскажу в главе 2;
- другие встроенные классы, структуры, интерфейсы, перечисления и делегаты.

Ранее я рассмотрел числовые и строковые типы, поддерживаемые языком C#. Все основные типы C# реализованы как классы или структуры в пространстве имен *System*. Например, тип данных *int*, является псевдонимом структуры *Int16*. Вместо определения переменной типа *int*:

```
int a = 55;
```

можно использовать:

```
System.Int16 a = 55;
```

Эти два оператора функционально идентичны. По этой же причине в одних случаях можно видеть строки C#, определенные так:

```
string str = "Hello, world!";
```

а в других — при определении строк тип данных *String* пишется с прописной буквы:

```
String str = "Hello, world!";
```

Использование и прописных, и строчных букв в этих операторах не означает, что C# иногда бывает нечувствительным к регистру. *String* с заглавной буквы —

это класс *String* пространства имен *System*. Если вы не указали оператор *using* для пространства имен *System* и хотите использовать *String* вместо *string*, придется ввести:

```
System.String str = "Hello, world!";
```

Вот типы C#, соответствующие классам и структурам пространства имен *System*:

**Псевдонимы типов данных C#**

| Со знаком | | Без знака | |
|---|---|---|---|
| **Тип .NET** | **Псевдоним C#** | **Тип .NET** | **Псевдоним C#** |
| *System.Object* | *object* | *System.Enum* | *enum* |
| *System.String* | *string* | *System.Char* | *char* |
| *System.SByte* | *sbyte* | *System.Byte* | *byte* |
| *System.Int16* | *short* | *System.UInt16* | *ushort* |
| *System.Int32* | *int* | *System.UInt32* | *uint* |
| *System.Int64* | *long* | *System.UInt64* | *ulong* |
| *System.Single* | *float* | *System.Double* | *double* |
| System.Decimal | decimal | System.Boolean | *bool* |

Поскольку базовые типы являются классами и структурами, у них могут быть поля, методы и свойства. Именно поэтому свойство *Length* можно использовать для получения количества символов в объекте *string*, а числовые типы данных имеют поля *MinValue* и *MaxValue*. Процедуры и методы, применяемые для поддержки массивов, реализованы в классе *System.Array*.

# Соглашения об именовании

Дальше я буду использовать соглашения об именовании, основанные на принципах .NET Framework и на системе, называемой венгерской нотацией в честь легендарного программиста Microsoft Чарльза Симони (Charles Simonyi).

Для определяемых мной имен классов, свойств и событий я буду применять *стиль языка Pascal* (Pascal casing): используются заглавные и строчные буквы, слово начинается с заглавной буквы, внутри слова также могут присутствовать заглавные буквы.

Для определяемых мной полей, переменных и объектов я буду применять *стиль верблюда* (camel casing): первая буква — строчная, но имя может содержать заглавные буквы (горбы).

Для переменных стандартных типов я буду использовать префикс, набранный строчными буквами и показывающий тип переменной. А префиксы такие:

| Тип данных | Префикс |
|---|---|
| *bool* | *b* |
| *byte* | *by* |
| *short* | *s* |
| *Int* | *i, x, y, cx, cy* |
| *long* | *I* |
| *float* | *f* |

(*продолжение*)

| Тип данных | Префикс |
|---|---|
| *char* | *ch* |
| *string* | *str* |
| *object* | *obj* |

Префиксы *x* и *y* означают, что переменная содержит координату точки, а *cx* и *cy* — что переменная содержит ширину или высоту (*c* — это сокращение от *count*.)

В именах объектов различных классов, я буду использовать префикс — имя класса, записанное строчными буквами, возможно, сокращенное. Например, объект типа *Point* может называться *ptOrigin*. Иногда в программах будет создаваться только один экземпляр данного класса. Тогда объект будет иметь то же имя, что и класс, но записанное строчными буквами. Например, объект типа *Form* будет называться *form*, а объект типа *PaintEventArgs — pea*.

У переменных-массивов перед остальными положенными им префиксами будет стоять префикс *a*.

# Выходим за рамки консоли

Осенью 1985 г. Microsoft выпустила первую версию Windows. Кроме того, тогда же она выпустила Windows Software Development Kit (SDK), который показал программистам, как писать приложения для Windows на языке C.

Первая программа «hello-world» в Windows 1.0 SDK оказалась немного скандальной. HELLO.C имела длину примерно 150 строк, также имелся файл ресурсов HELLO.RC длиной не менее 20 строк. Надо заметить, что программа создавала меню и показывала диалоговое окно, но даже без этих удобств она содержала 70 строк кода. У ветеранов программирования на C первое знакомство с программой «hello-world» для Windows вызывало смех или ужас.

В известном смысле вся история новых языков программирования и библиотек классов для Windows связана с борьбой за то, чтобы сделать программу «hello-world» для Windows небольшой, простой и элегантной.

Давайте посмотрим, оправдают ли Windows Forms эти ожидания.