ВЕДЬ ЭТО ТАК ПРОСТО!

C#

для ЧАЙНИКОВ®

Издательство ДИАЛЕКТИКА

Быстрое создание приложений

Применение интерфейсов и объектно-ориентированных концепций

Коллекции, потоки, события и многое другое

Джон Пол Мюллер
при участии **Билла Семпфа и Чака Сфера**

C#

для чайников®

C# 7.0

ALL - IN - ONE

by John Paul Mueller,
Bill Sempf, and Chuck Sphar

for dummies®
A Wiley Brand

C#

Джон Пол Мюллер
при участии
Билла Семпфа и
Чака Сфера

для чайников®

Диалектика

Москва ◆ Санкт-Петербург
2019

ББК 32.973.26-018.2.75
М98
УДК 681.3.07

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Мюллер, Джон Пол, **Семпф**, Билл, **Сфер**, Чак.
М98 C# для чайников. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 608 с. : ил. — Парал. тит. англ.

ISBN 978-5-907144-43-9 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Джон Пол Мюллер, Билл Семпф, Чак Сфер

C# для чайников

Подписано в печать 04.09.2019. Формат 70x100/16
Усл. печ. л. 38,0. Уч.-изд. л. 31,3
Доп. тираж 500 экз. Заказ № 7145

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-43-9 (рус.)
ISBN 978-111-9-42811-4 (англ.)

Оглавление

Содержание

Об авторе

Джон Мюллер — независимый автор и технический редактор. На сегодняшний день он написал 104 книги и более 600 статей на самые разные темы: от сетей до искусственного интеллекта и от управления базами данных до головокружительного программирования. Некоторые из его текущих работ включают книгу о машинном обучении, пару книг по Python и книгу о MATLAB. Благодаря навыкам технического редактора Джон помог более чем 70 авторам усовершенствовать свои рукописи. Джон всегда интересовался разработкой программного обеспечения и писал о самых разных языках, включая очень успешную книгу по языку программирования C++. Обязательно прочитайте блог Джона по адресу `http://blog.johnmuellerbooks.com/`. Связаться с ним можно по адресу `John@JohnMuellerBooks.com`.

Посвящение

Ребекке. Ты навечно в моем сердце!

Благодарности

Спасибо моей жене, Ребекке. Несмотря на то что она покинула этот мир, ее дух есть в каждой книге, которую я пишу, и в каждом слове, которое появляется на странице. Она верила в меня, когда не верил никто.

Расс Маллен (Russ Mullen) заслуживает благодарности за техническое редактирование этой книги. Он существенно усилил точность и глубину изложения материала. Расс работал исключительно усердно, помогая в поиске ошибок, находя нужные материалы, а также внося множество предложений. Эта книга особенно трудна из-за изменений, связанных с переходом на C# 7.0 и Visual Studio 2017. Пришлось принимать много действительно сложных решений, в чем Расс мне очень помог.

Мой агент Мэтт Вагнер (Matt Wagner) заслуживает похвалы за то, что помог мне получить контракт и позаботился обо всех деталях, о которых большинство авторов постоянно забывают. Я очень ценю его помощь.

Наконец, я хотел бы поблагодарить Кэти Мор (Katie Mohr), Сьюзен Кристоферсен (Susan Christophersen) и других сотрудников редакции John Wiley & Sons, Inc. за их беспрецедентную поддержку этой работы.

Введение

C# — удивительный язык! Можно использовать один лишь этот язык для всего — от разработки настольных приложений до создания веб-приложений и даже веб-интерфейсов прикладного программирования (API). В то время как другие разработчики вынуждены преодолевать недостатки своих языков для создания даже небольшого подмножества видов приложений, которые легко поддерживает C#, вы можете быстро, не прикладывая ни физических, ни умственных усилий, кодировать свои приложения, тестировать их, а затем наслаждаться бездельем, сидя на пляже. Конечно, любой язык, который на такое способен, необходимо изучать, и в этом вам поможет наша книга, которая представляет собой дверь в новые возможности в программировании.

Вам интересно, почему нужна именно эта книга, а не какая-то иная? Да потому что она обучает основам языка C#, которые все-таки желательно знать, прежде чем начать его использовать. Здесь изложены основы языка программирования C#, и вы сможете сразу же приступить к работе с интегрированной средой разработки Visual Studio 2017. Из этой книги вы получите максимум информации о C# 7.0 за минимальные сроки.

Об этой книге

Даже если у вас есть опыт работы с C#, новые функциональные возможности C# 7.0 помогут вам создавать многофункциональные приложения еще быстрее, чем ранее. В этой книге представлено множество новых функциональных возможностей данного языка программирования. Например, вы обнаружите новые методы проверки соответствия текста шаблонам, которые предоставляет C# 7.0. Вы также откроете для себя чудеса использования кортежей и локальных функций. Усовершенствовано даже использование литералов, но вам придется заглянуть внутрь книги, чтобы узнать, как именно. Эта книга предназначена для того, чтобы вы могли освоить C# 7.0 быстро и легко; в ней максимально устранены все сложности, которые могут возникнуть при попытке изучить эти темы онлайн.

Чтобы помочь вам усваивать концепции книги, в ней используются следующие соглашения.

- Текст, который нужно вводить в том же виде, в каком он приведен в книге, **выделен моноширинным полужирным шрифтом**.
- Слова, которые следует ввести, могут быть также выделены *курсивом*; в этом случае они используются в качестве местозаполнителей. Это означает, что вам нужно заменить их чем-то конкретным. Например, если вы видите инструкцию "Введите ***свое имя*** и нажмите клавишу <Enter>", вам нужно заменить *свое имя* своим реальным именем.
- *Курсив* использован также для определяемых терминов. Это означает, что вы не должны полагаться на другие источники информации, чтобы получить разъяснение необходимой терминологии.
- Адреса в вебе даны моноширинным шрифтом.
- Когда нужно будет с помощью мыши выполнить некоторую последовательность команд, они будут разделены специальной стрелкой наподобие этой: File⇨New File. Эта запись гласит, что вы должны щелкнуть мышью сначала на пункте меню File, а затем — на New File.

Глупые предположения

Вам, может быть, трудно в это поверить, но мы не делаем почти никаких предположений о своем читателе. В конце концов, мы с вами даже не знакомы! Но для того, чтобы от чего-то отталкиваться, нам все же приходится хоть как-то вас представлять.

Наиболее важное предположение — что вы знаете, как использовать Windows, на вашем компьютере есть правильно установленная копия Windows и вы умеете работать с приложениями Windows. Если установка приложений для вас — темный лес, скорее всего, вам будет трудно использовать эту книгу. При чтении книги вам придется устанавливать некоторые приложения, выяснять, как их использовать, и создавать собственные простые приложения самостоятельно.

Вам также в некоторой степени нужно знать, как работать с Интернетом. Многие из материалов, включая загружаемые исходные тексты, находятся в Интернете, и вам желательно загрузить их оттуда, чтобы получить максимальную выгоду от работы с книгой.

Пиктограммы, используемые в книге

По мере чтения этой книги вы будете сталкиваться с пиктограммами, которыми отмечен важный материал. Вот что они означают.

СОВЕТ

Советы хороши тем, что помогают сэкономить время и выполнить некоторую задачу без излишних усилий. Советы в этой книге представляют собой методы работы, позволяющие сэкономить время, или указатели на ресурсы, которые стоит использовать, чтобы получить максимальную выгоду от C#.

ВНИМАНИЕ!

Не хочется строить из себя рассерженных родителей, но вы и в самом деле должны избегать всего, что помечено такой пиктограммой. В противном случае может обнаружиться, что ваше приложение не работает, как ожидалось, что вы получаете неправильные ответы от внешне идеальных алгоритмов или что (в худшем случае) вы потеряли свои данные.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Всякий раз, увидев этот значок, знайте, что вас ждет расширенный совет или новая методика. Вы можете счесть эти заметки слишком скучными, так что, если хотите, можете их игнорировать.

ЗАПОМНИ!

Обратите особое внимание на материал, отмеченный этой пиктограммой. Этот текст обычно содержит важную информацию, которую необходимо знать для работы с C#.

Источники дополнительной информации

На этой книге ваше обучение языку программирования C# не заканчивается — на самом деле оно только начинается. Джон Мюллер (John Mueller) предоставил ряд материалов в вебе, чтобы лучше удовлетворить ваши потребности. Вы также можете написать ему электронное письмо. Он поможет решить ваши вопросы, связанные с книгой, и расскажет об обновлениях C# и других материалах, связанных с книгой, в своем блоге. Вот к каким материалам вы получаете доступ.

» **Шпаргалка**. Вы пользовались шпаргалками в школе? Вот такую же шпаргалку мы вам здесь предоставляем. В ней содержатся некоторые заметки о задачах, которые вы можете решать с помощью C# и

которые знает не каждый человек. Чтобы найти шпаргалку для этой книги, перейдите на сайт `www.dummies.com` и поищите *C# 7.0 All-in-One For Dummies Cheat Sheet*. В ней содержится самая разная информация, связанная с C#.

» **Обновления**. Изменения иногда случаются. Например, мы могли просмотреть предстоящие изменения, когда использовали свой хрустальный шар во время написания этой книги. В прошлом это означало, что книга стала устаревшей и менее полезной, но сейчас вы можете найти ее обновления на сайте по адресу `www.dummies.com`. Помимо этих обновлений, можно просмотреть блог с ответами на вопросы читателей и демонстрацией полезных методов работы, связанных с книгой, по адресу `http://blog.johnmuellerbooks.com/`.

» **Сопутствующие файлы**. Вы действительно собираетесь вводить весь представленный в книге код вручную? Большинство читателей предпочитают тратить свое время на более интересные занятия. К счастью для них и для вас, примеры, используемые в книге, доступны для загрузки из веба, так что все, что вам нужно сделать, — это прочитать книгу, чтобы познакомиться с методами разработки программ на C#. Все файлы с исходными текстами вы можете найти на сайте `www.dummies.com` на вкладке загрузок.

Что дальше

Любой, кто не знаком с C#, должен начать с главы 1, "Ваше первое консольное приложение C#", и последовательно идти до конца книги. Эта книга призвана с самого начала облегчить для вас открытие преимуществ использования C#. Позднее, после того как вы уже увидите достаточное количество кода C#, можно будет установить Visual Studio и попробовать работать с примерами программ из данной книги.

В целом, чем больше вы знаете о C#, тем с более позднего места книги можете начать ее чтение.

В книге предполагается, что вы с самого начала хотите видеть код C#. Однако, если вы хотите взаимодействовать с этим кодом, вам нужно иметь установленную копию Visual Studio 2017 (некоторые из примеров в книге не будут работать с более старыми версиями Visual Studio). Чтобы гарантировать, что каждый сможет это сделать, книга ориентирована на пакет Visual Studio 2017 Community Edition, который распространяется бесплатно. Вы можете открыть для себя чудеса C# 7.0, не платя ни копейки!

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:
E-mail: `info@dialektika.com`
WWW: `http://www.dialektika.com`

Основы программирования на C#

В ЭТОЙ ЧАСТИ...

- Глава 1, "Ваше первое консольное приложение на C#"
- Глава 2, "Работа с переменными"
- Глава 3, "Работа со строками"
- Глава 4, "Операторы"
- Глава 5, "Управление потоком выполнения"
- Глава 6, "Глава для коллекционеров"
- Глава 7, "Работа с коллекциями"
- Глава 8, "Обобщенность"
- Глава 9, "Эти исключительные исключения"
- Глава 10, "Списки элементов с использованием перечислений"

Глава 1

Ваше первое консольное приложение на C#

В ЭТОЙ ГЛАВЕ...

- Краткое введение в программирование
- Создание простого консольного приложения
- Изучение консольного приложения
- Сохранение кода для последующей работы

В этой главе вы бегло ознакомитесь с компьютерами и компьютерными языками, включая компьютерный язык C# (произносится как "си-шарп") и Visual Studio 2017. Затем вы создадите простую программу, написанную на языке C#.

Компьютерные языки, C# и .NET

Компьютер — удивительно быстрый, но невероятно тупой помощник человека. Компьютеры делают то, о чем их просят (в разумных пределах, конечно), и делают все очень быстро (и чем дальше — тем быстрее).

К сожалению, компьютеры не понимают ничего похожего на человеческий язык. Да, вы можете удивленно сказать мне: "Неправда, мой телефон позволяет мне позвонить другу, просто произнеся его имя". Да, в телефоне работает крошечный компьютер. Получается, что компьютер владеет человеческой речью? Нет, речь понимает компьютерная *программа*, а не сам компьютер.

Язык, который действительно понимают компьютеры, — это *машинный язык*. Человек, в принципе, может написать нечто на машинном языке, но это чрезвычайно трудная и подверженная ошибкам задача.

Люди и компьютеры встречаются где-то посредине. Программисты создают программы на языке, который не столь гибкий, как человеческая речь, но намного более гибкий и простой в использовании по сравнению с машинным языком. Языки, занимающие эту средину, в частности C#, — это компьютерные языки *высокого уровня*. (*Высокий* в данном случае является относительным термином.)

Что такое программа

Что такое программа? В практическом смысле программа Windows представляет собой выполнимый файл, который можно запустить на выполнение, дважды щелкнув на его пиктограмме. Например, редактор Microsoft Word, используемый для написания этой книги, — это *выполнимая программа* (executable program). Имена выполнимых файлов обычно заканчиваются расширением `.exe`. Например, выполнимый файл Microsoft Word — `Winword.exe`.

Но программа представляет собой и нечто иное. Выполнимая программа состоит из одного или нескольких *исходных файлов* (source files). Исходный файл C# представляет собой текстовый файл, содержащий последовательность команд C#, которые сочетаются по законам грамматики языка C#. Этот файл называется исходным, вероятно, потому, что у начинающего программиста создается впечатление, что из него проистекают разочарования и тревоги.

Ах, да, еще грамматика? Да, грамматика, но грамматика особого C#-рода, которая гораздо проще той, с которой большинство людей боролось в средней школе.

Что такое C#

Язык программирования C# является одним из промежуточных языков, используемых программистами для создания выполнимых программ. C# сочетает в себе мощь языка программирования C++ (произносится как "си плюс плюс") с простотой Visual Basic. (Новейшие .NET-воплощения Visual Basic во многих отношениях практически равны C#. Как флагманский язык .NET C#, как правило, первым получает большинство новых функциональных возможностей.) Файл программы на языке C# имеет расширение `.cs`.

C# представляет собой язык, обладающий рядом свойств.

- **Гибкость**. Программы на C# могут выполняться и на вашей локальной машине, и быть переданы через Интернет и выполняться на некотором удаленном компьютере.

- **Мощь**. C# имеет, по сути, тот же набор команд, что и C++, но "со сглаженными острыми углами".
- **Простота в использовании**. Защита от ошибок в C# спасает ваш код от большинства ошибок C++, поэтому вы тратите гораздо меньше времени на отладку своих программ.
- **Визуальная ориентированность**. Библиотека кода .NET, которую C# использует для реализации многих своих возможностей, предоставляет помощь, необходимую для легкого создания сложных окон с раскрывающимися списками, вкладками, сгруппированными кнопками, полосами прокрутки и фоновыми изображениями.


СОВЕТ

.NET произносится как "дот нет".

- **Дружественность к Интернету**. C# играет ключевую роль в платформе .NET Framework, нынешнем подходе Microsoft к программированию для Windows, Интернета и за их пределами.
- **Безопасность**. Любой язык, предназначенный для использования в Интернете, должен включать серьезные средства защиты от злонамеренных хакеров.

Наконец, C# является неотъемлемой частью .NET.


ЗАПОМНИ!

Эта книга, в первую очередь, посвящена языку C#. Если ваша основная цель — использовать Visual Studio, программировать для Windows 8 или 10 либо ASP.NET, то вместе с данной книгой рекомендуем приобрести книги серии ...*для чайников* по этим темам.

Что такое .NET

Проект .NET начался в 2002 году как стратегия Microsoft, призванная открыть веб для простых смертных вроде нас с вами. На сегодняшний день это больше, чем все, что делает Microsoft. В частности, это новый способ программирования для Windows. Он также дает основанный на C язык программирования C# и простые визуальные инструменты, сделавшие настолько популярным Visual Basic.

Небольшой экскурс в данную тему поможет вам увидеть корни C# и .NET. Интернет-программирование в более старых языках, таких как C и C++, традиционно было очень сложным. Sun Microsystems ответили на эту проблему, создав язык программирования Java. Чтобы создать Java, Sun взяла грамматику C++, сделала ее более удобной для пользователя и сосредоточила ее на разработке распределенных приложений.

ЗАПОМНИ!

Когда программисты говорят "распределенный", они имеют в виду географически разбросанные компьютеры, на которых работают программы, общающиеся между собой (в большинстве случаев — через Интернет).

Когда несколько лет назад компания Microsoft лицензировала Java, она столкнулась с юридическими трудностями с Sun по поводу изменений, которые она хотела внести в язык. В результате Microsoft более или менее отказалась от Java и начала искать способы конкурировать с этим языком программирования.

Такое вытеснение Microsoft из Java пошло на пользу, потому что у Java есть серьезная проблема: хотя Java — язык со множеством возможностей, чтобы получить полную выгоду от его применения, следует писать *на Java* всю программу полностью. У Microsoft было слишком много разработчиков и слишком много миллионов строк исходного кода, так что ей пришлось придумывать способ каким-то образом поддерживать несколько языков. Так возникла концепция .NET.

.NET — это каркас, во многом похожий на библиотеки Java (а язык C# очень похож на язык Java). Так же, как Java представляет собой и сам язык, и обширную библиотеку кода, C# в действительности представляет собой нечто намного большее, чем просто ключевые слова и синтаксис языка C#. Это еще и все, чем обладает хорошо организованная библиотека, содержащая тысячи элементов кода, которые упрощают выполнение любого вида программирования, которое вы можете себе представить, — от веб-баз данных до криптографии или даже до скромного диалогового окна Windows.

Microsoft заявила, что .NET намного превосходит набор веб-инструментов Sun, основанный на Java, но это не главное. В отличие от Java, .NET не требует переписывания существующих программ. Программист на Visual Basic может добавить всего лишь несколько строк, чтобы сделать существующую программу работающей через Интернет (т.е. такая программа знает, как получать необходимые данные из Интернета). .NET поддерживает все распространенные языки Microsoft — и сотни других языков, написанных сторонними поставщиками. Однако флагманским языком флота .NET является C#. Именно он оказывается первым языком, обеспечивающим доступ к каждой новой функциональной возможности .NET.

Что такое Visual Studio 2017 и Visual C#

Первым "визуальным" языком от Microsoft был Visual Basic. Первым популярным языком от Microsoft на базе C был Visual C++. Как и Visual Basic, он имел в названии слово "Visual", поскольку имел встроенный графический интерфейс пользователя (graphical user interface — GUI). В этом графическом интерфейсе было все, что нужно для разработки отличных программ на C++.

В конечном итоге Microsoft перевела все свои языки в единую среду — Visual Studio. Распробовав Visual Studio 6.0, разработчики с нетерпением ожидали появления версии 7. Однако вскоре перед ее выпуском Microsoft решила переименовать ее в "Visual Studio .NET", чтобы подчеркнуть связь новой среды с .NET.

Для большинства это звучало как маркетинговая уловка, пока они не начали в нее вникать. Visual Studio .NET в действительности достаточно отличалась от своих предшественников, чтобы оправдать новое имя. Visual Studio 2017 является потомком в девятом поколении оригинальной версии Visual Studio .NET.


ЗАПОМНИ!

Microsoft называет свою реализацию языка "Visual C#". На самом деле Visual C# является ни чем иным, как компонентом C# в составе Visual Studio. C# — это C# в составе Visual Studio или без него. Теоретически вы можете писать программы на C# с помощью любого текстового редактора и нескольких специальных инструментов, но работать с помощью Visual Studio гораздо проще.

Создание первого консольного приложения

Visual Studio 2017 включает в себя мастер приложений, который создает шаблонные программы и сокращает много грязной работы, которую вам пришлось бы делать самостоятельно, если бы вы делали все сами "с нуля". (Самое меньшее, что можно сказать о стремлении все делать самому "с нуля", — это чревато ошибками.)

Как правило, такие шаблонные программы в действительности ничего не делают, по крайней мере ничего полезного. Тем не менее они позволяют преодолеть начальные препятствия — те, которые возникают еще до начала работы. Некоторые из стартовых программ достаточно сложны (на самом деле вы будете удивлены тем, сколько возможностей предоставляет мастер приложений, особенно для графических программ).

Однако рассматриваемая здесь стартовая программа не является графической. *Консольное* приложение — это приложение, которое запускается на "консоли" Windows, обычно именуемой приглашением DOS или командным окном. Если вы нажмете <Ctrl+R>, а затем введете `cmd`, то увидите командное окно. Это и есть консоль, на которой будет запущено приложение.


ЗАПОМНИ!

Приведенные далее инструкции относятся к Visual Studio. Если вы используете что-либо, отличное от Visual Studio, обратитесь к документации, прилагаемой к вашей среде. Кроме того, вы можете просто ввести исходный код непосредственно в среду C#.

Создание исходной программы

Чтобы запустить Visual Studio, нажмите кнопку <Windows> на клавиатуре и введите **`Visual Studio`**. Одним из возможных вариантов является Visual Studio 2017. Вы можете получить доступ к примеру кода для этой главы в папке `\CSAIO4D\BK01\CH01` из описанного во введении загружаемого источника.

Выполните следующие действия для создания вашего консольного приложения C#.

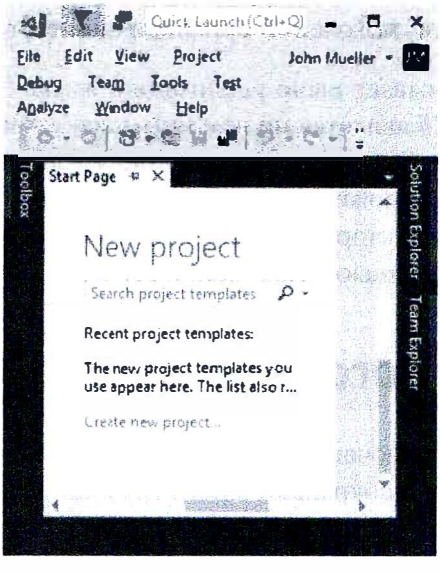


Рис. 1.1. Окно создания нового проекта помогает вам в создании лучшего приложения для Windows

1. **Откройте Visual Studio 2017 и щелкните на пункте создания нового проекта Create New Project, показанном на рис. 1.1.**

   Visual Studio покажет вам ряд пиктограмм, представляющих различные типы приложений, которые вы можете создать (рис. 1.2).

2. **В окне нового проекта New Project щелкните на пиктограмме консольного приложения Console App (.NET Framework).**

ВНИМАНИЕ!

Убедитесь, что на панели типов Project Types проектов вы выбрали Visual C#, а под ним — Windows; в противном случае Visual Studio может создать нечто ужасное, например приложение Visual Basic или Visual C++. Затем на панели шаблонов Templates щелкните на пиктограмме консольного приложения Console App (.NET Framework).

ЗАПОМНИ!

Visual Studio требует, чтобы перед тем, как начать вводить свою программу на C#, вы создали проект. *Проект* — это папка, в которую вы помещаете все файлы, входящие в вашу программу. Он имеет также набор конфигурационных файлов, которые помогают компилятору выполнять свою работу.

Когда вы говорите своему компилятору, что нужно *скомпилировать* программу, он упорядочивает содержимое проекта, чтобы найти файлы, необходимые для создания выполнимой программы.

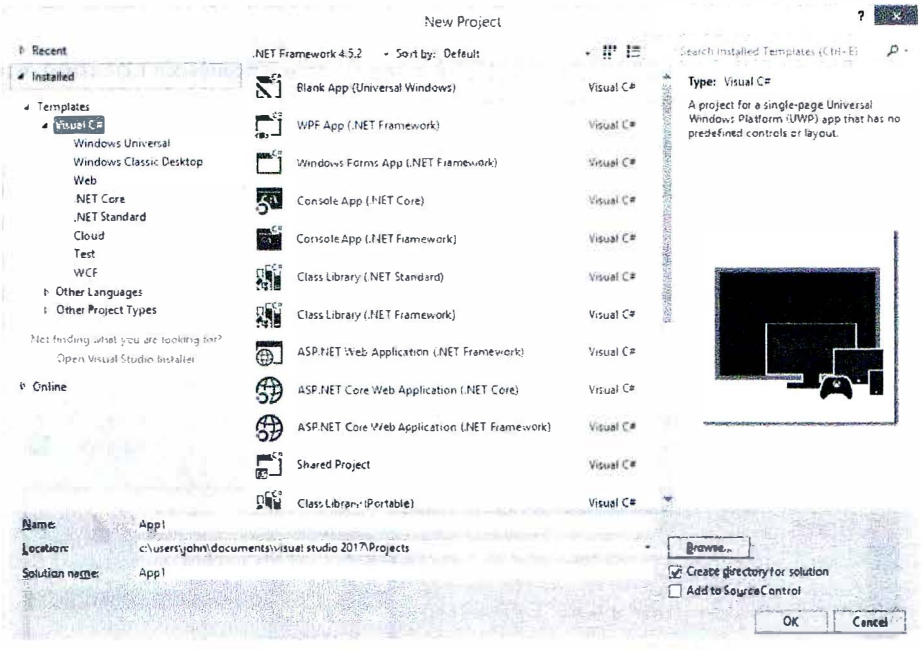


Рис. 1.2. Мастер приложений Visual Studio создает для вас новую программу

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Visual Studio 2017 предоставляет поддержку как приложений .NET Framework, так и .NET Core. Приложение .NET Framework представляет собой то же самое, что и приложения C#, поддерживаемые в предыдущих версиях Windows; оно работает только в Windows и не является приложением с открытым исходным кодом. Приложение .NET Core может работать в операционных системах Windows, Linux и Mac и полагается на установку с открытым исходным кодом. Хотя использование .NET Core может показаться идеальным, приложения .NET Core поддерживают только часть функций .NET Framework, и вы не сможете добавить к ним графический интерфейс. Microsoft создала .NET Core для следующих целей.

- Кроссплатформенная разработка
- Микрослужбы
- Докерные контейнеры
- Высокопроизводительные и масштабируемые приложения
- Поддержка сторонних приложений .NET

3. **Имя по умолчанию вашего первого приложения — App1, но в этот раз измените его на `Program1`, введя данное имя в поле имени Name.**

СОВЕТ

По умолчанию место хранения проекта находится где-то в глубине вашего каталога документов `Documents`. Для большинства разработчиков гораздо лучше размещать файлы там, где можно их найти и взаимодействовать с ними по мере необходимости, а не обязательно там, где их хочет разместить Visual Studio.

4. **Введите `C:\CSAIO4D\BK01\CH01` в поле местоположения Location, чтобы изменить расположение файлов проекта.**

5. **Щелкните на кнопке OK.**

   После небольшого жужжания и щелканья винчестера Visual Studio создает файл с именем `Program.cs`. (Если вы посмотрите в окно обозревателя решений Solution Explorer, показанное на рис. 1.3, то увидите некоторые другие файлы; пока что просто игнорируйте их.) Если обозреватель решений не отображается, выберите в меню View⇨Solution Explorer (Вид⇨Обозреватель решений).

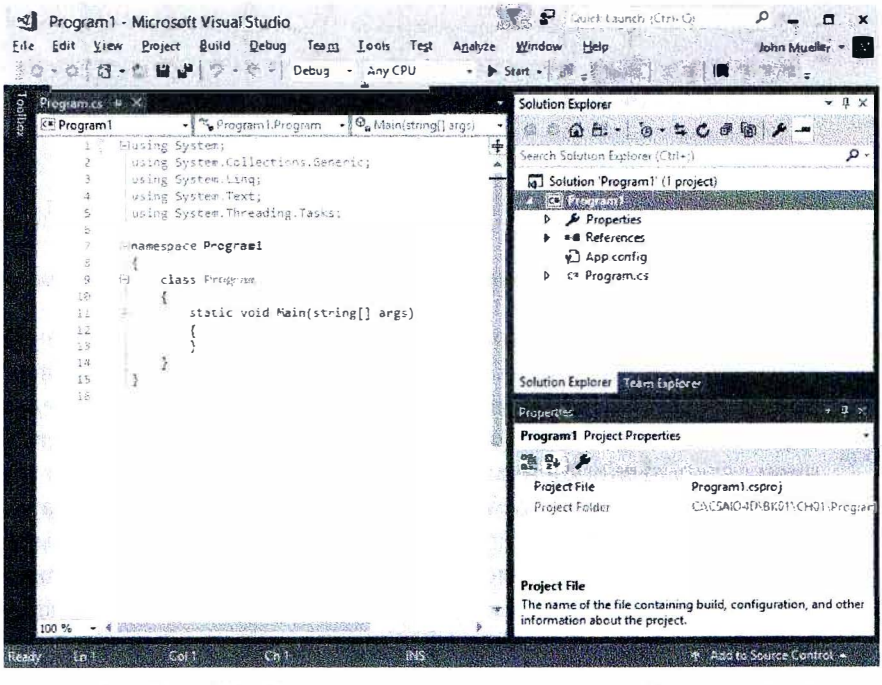


Рис. 1.3. Visual Studio показывает только что созданный проект

Исходные файлы C# имеют расширение `.cs`. Имя `Program` представляет собой имя по умолчанию, присваиваемое файлу программы.

Вот как выглядит содержимое вашего первого консольного приложения (рис. 1.3):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Program1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

СОВЕТ

Вы можете вручную изменить местоположение каждого проекта. Однако имеется более простой способ. Работая с книгой, вы можете изменить местоположение программ по умолчанию. Чтобы это сделать, выполните следующие действия после завершения создания проекта.

1. **Выберите пункт меню Tools⇨Options (Средства⇨Параметры).**

   Откроется диалоговое окно Options (Параметры). Вы можете выбрать в нем флаг показа всех параметров Show All Options.

2. **Выберите Projects and Solutions⇨General (Проекты и решения⇨Общие).**

3. **Выберите новое местоположение для своих файлов и щелкните на кнопке OK.**

   (В примерах данной книги предполагается, что вы используете в качестве местоположения по умолчанию каталог `C:\CSAIO4D`.)

Указанное диалоговое окно можно увидеть на рис. 1.4. Пока что не трогайте другие поля в настройках проекта.

ЗАПОМНИ!

Вдоль левого края окна кода видны несколько маленьких плюсов (+) и минусов (-) в квадратиках. Нажмите знак + рядом с `using...` При этом будет развернута соответствующая область кода. Это удобная функциональная возможность Visual Studio, которая минимизирует беспорядок на экране. Вот директивы, которые появляются при развертывании области в консольном приложении по умолчанию:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

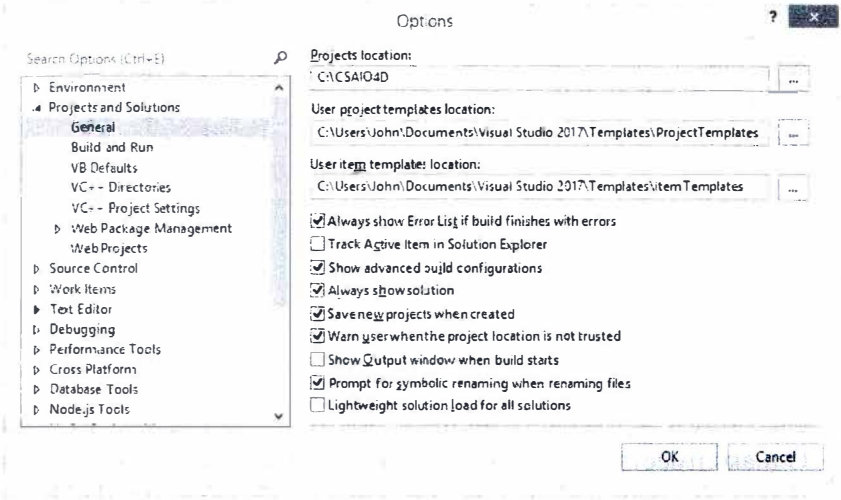


Рис. 1.4. Изменение расположения проекта по умолчанию

Регионы (region) помогают программисту сосредоточиться на коде, над которым вы работаете, скрывая код, который в настоящий момент вас не интересует. Некоторые блоки кода, такие как блок `namespace`, блок `class`, методы и другие элементы кода, автоматически получают +/- без директивы `#region`. При желании вы можете добавить свои сворачиваемые регионы, набрав `#region` перед разделом кода и `#endregion` — после него. Кроме того, вы можете указать название региона, например `открытые методы`. Такой раздел кода выглядит следующим образом:

```
#region Открытые методы
... Ваш код
#endregion Открытые методы
```

ЗАПОМНИ!

Это имя может содержать пробелы. Можно также вкладывать один регион в другой, но перекрываться регионы не могут.

В настоящий момент `using System;` является единственной директивой `using`, которая нам необходима. Остальные директивы можно удалить; компилятор даст вам знать, если вы удалите лишнее.

Тестовая поездка

Прежде чем пытаться создать приложение, откройте окно Output (Вывод) (если оно еще не открыто), воспользовавшись пунктом меню View⇨Output (Вид⇨Вывод). Чтобы преобразовать исходный текст C# в выполнимую программу, выберите Build⇨Build Program1 (Сборка⇨Построить Program1). Visual Studio

ответит следующим сообщением (при другой локализации Visual Studio оно может быть другим):

```
------ Сборка начата: Project: Program1, Configuration: Debug Any CPU ---
Program1 -> C:\CSAIO4D\BK01\CH01\Program1\Program1\bin\Debug\Program1.exe
=== Сборка: успешно: 1, с ошибками: 0, без изменений: 0, пропущено: 0 ===
```

Ключевой является часть последней строки, гласящая `успешно: 1`.

СОВЕТ

В общем случае в программировании `succeeded` (`успешно`) означает "хорошо", ну а `failed` (`с ошибками`) — "плохо".

Чтобы выполнить программу, выберите пункт меню Debug⇨Start (Отладка⇨Запуск). Программа выведет черное окно консоли и немедленно завершится. (Если у вас быстрый компьютер, внешний вид этого окна — мелькание чего-то темного на экране.) Программа, похоже, ничего не сделала. И это на самом деле так. Шаблон — это ни что иное, как пустая оболочка.

СОВЕТ

Альтернативная команда, Debug⇨Start Without Debugging (Отладка⇨Запуск без отладки), ведет себя немного лучше. Испробуйте ее.

Заставим программу работать

Отредактируйте шаблон `Program.cs` так, чтобы он имел следующий вид:

```
using System;
namespace Program1
{
    public class Program
    {
        // Здесь программа начинает работу.
        static void Main(string[] args)
        {
            // Приглашение пользователю ввести свое имя.
            Console.WriteLine("Введите ваше имя:");

            // Чтение введенного имени.
            string name = Console.ReadLine();

            // Приветствие пользователя по имени.
            Console.WriteLine("Привет, " + name);

            // Ожидание реакции пользователя.
            Console.WriteLine("Нажмите <Enter> для выхода...");
            Console.Read();
        }
    }
}
```

СОВЕТ

Не беспокойтесь из-за двойных или тройных косых черт (// или ///) и о том, нужно ли вводить один или два пробела или одну или две новые строки. Однако обратите внимание на большие буквы.

Выберите Build⇨Build Program1 (Сборка⇨Построить Program1), чтобы преобразовать эту новую версию `Program.cs` в программу `Program1.exe`.

В Visual Studio 2017 выберите в меню Debug⇨Start Without Debugging (Отладка⇨Запуск без отладки). Появится черное окно консоли, в котором вам предложат ввести свое имя. (Возможно, вам нужно будет активировать окно консоли, щелкнув на нем.) Затем в окне отобразится слово `Привет`, введенное имя и строка `Нажмите <Enter> для выхода...` Нажатие клавиши <Enter> закрывает окно.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Вы также можете выполнить программу из командной строки DOS. Для этого откройте окно командной строки и введите следующее:

```
CD \C#Programs\Program1\bin\Debug
```

Теперь введите **`Program1`** для выполнения программы. Вывод программы должен выглядеть так же, как и ранее. Вы также можете перейти к папке `\C#Programs\Program1\bin\Debug` в проводнике Windows, а затем выполнить двойной щелчок на файле `Program1.exe`.

СОВЕТ

Чтобы открыть окно командной строки, попробуйте воспользоваться командой меню Tools⇨Command Prompt (Средства⇨Командная строка). Если такая команда в вашем Visual Studio недоступна, откройте проводник Windows, найдите папку, содержащую выполнимый файл, как показано на рис. 1.5, и выберите пункт меню File⇨Open Command Prompt (Файл⇨Открыть командную строку). Вы увидите окно с приглашением командной строки, в котором сможете выполнить свою программу.

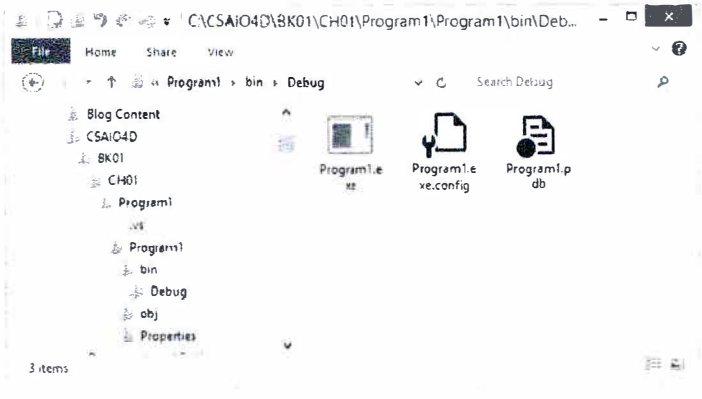


Рис. 1.5. Проводник Windows позволяет легко открыть окно командной строки

Обзор консольного приложения

В следующих разделах мы рассмотрим поочередно все части вашего первого консольного приложения C#, чтобы разобраться, как оно работает.

Каркас программы

Основной каркас всех консольных приложений имеет следующий вид:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Program1
{
    public class Program
    {
        // Здесь программа начинает работу.
        public static void Main(string[] args)
        {
            // Здесь находится ваш код.
        }
    }
}
```

Программа начинает выполняться с инструкции, находящейся после `Main()`, и заканчивается закрытой фигурной скобкой (`}`), следующей за `Main()`. (Со временем вы получите все необходимые пояснения. Пока просто знайте, что все работает именно так.)

ЗАПОМНИ!

Список директив `using` может находиться непосредственно перед фразой `namespace Program1 {` или сразу после нее. Порядок директив не имеет значения. Вы можете применить `using` к большому количеству сущностей в .NET. О том, что означают пространства имен и директивы `using`, вы узнаете из глав об объектно-ориентированном программировании в части 2, "Объектно-ориентированное программирование на C#".

Комментарии

Шаблон включает некоторые строки (а код примера содержит еще несколько таких дополнительных строк), такие как показанная ниже полужирным шрифтом:

```
// Здесь программа начинает работу.
public static void Main(string[] args)
```

C# игнорирует первую строку этого примера, которая известна как *комментарий*.

СОВЕТ

Любая строка, начинающаяся с // или ///, представляет собой просто текст, который C# игнорирует. Пока что рассматривайте // и /// как эквивалентные записи.

Почему нужно включать в программу строки, которые компьютер игнорирует? Потому что комментарии объясняют ваши инструкции C#. Программу, даже на C#, понять нелегко. Помните, что язык программирования является компромиссом между тем, что понимают компьютеры, и тем, что понимают люди. Эти комментарии полезны, когда вы пишете код, но особенно полезны они бедному программисту — возможно, вам самому, — который пытается восстановить вашу логику спустя год после написания. Комментарии существенно облегчают работу над программой.

СОВЕТ

Комментируйте почаще и попонятнее.

Тело программы

Реальная суть программы заключена в пределах блока кода внутри `Main()`, как показано далее:

```
// Приглашение пользователю ввести свое имя.
Console.WriteLine("Введите ваше имя:");

// Чтение введенного имени.
string name = Console.ReadLine();

// Приветствие пользователя по имени.
Console.WriteLine("Привет, " + name);
```

СОВЕТ

Вы можете здорово сэкономить на вводе исходного текста с помощью функциональной возможности фрагментов кода C# (C# Code Snippets). Фрагменты отлично подходят для обычных инструкций, таких как `Console.WriteLine`. Нажмите <Ctrl+K,X>, чтобы увидеть всплывающее меню фрагментов. (Возможно, вам нужно будет нажать один или два раза <Tab>, чтобы открыть папку Visual C# или другие папки в этом меню.) Прокрутите меню до `cw` и нажмите <Enter>. Visual Studio вставит тело готовой к работе инструкции `Console.WriteLine()` с точкой вставки между круглыми скобками. Если у вас есть несколько ярлыков, таких как `cw`, `for` или `if`, используйте еще более быструю технику: введите **`cw`** и дважды нажмите <Tab>. (Попробуйте также выделить несколько строк кода, нажмите <Ctrl+K>, а затем нажмите <Ctrl+S>. Выберите что-то наподобие `if`. Конструкция `if` будет *окружать* выделенные строки кода.)

Программа начинает выполнение с первой инструкции C#: `Console.WriteLine`. Эта команда записывает строку символов `"Введите ваше имя:"` в консоль.

Следующая инструкция читает ответ пользователя и сохраняет его в *переменной* (в своего рода ящике для значений) с именем `name`. (См. главу 2, "Работа с переменными", для получения дополнительной информации об этих местах хранения.) Последняя строка объединяет строку `"Привет,"` с именем пользователя и выводит результат на консоль.

Последние три строки заставляют компьютер ждать, пока пользователь нажмет <Enter> для продолжения. Эти строки гарантируют, что у пользователя будет время для чтения вывода до продолжения программы:

```
// Ожидание реакции пользователя.
Console.WriteLine("Нажмите <Enter> для выхода...");
Console.Read();
```

Этот шаг может быть важным в зависимости от того, как вы выполняете программу, и в зависимости от используемой среды. В частности, запуск консольного приложения внутри Visual Studio или из проводника Windows делает предыдущие строки необходимыми; в противном случае окно консоли закрывается так быстро, что вы не успеваете прочитать результат. Если вы откроете окно консоли и запустите программу из него, окно останется открытым независимо от того, работает ли программа.

Введение в хитрости панели элементов

Ключевая часть программы, созданной в предыдущем разделе, состоит из последних двух строк кода:

```
// Ожидание реакции пользователя.
Console.WriteLine("Нажмите <Enter> для выхода...");
Console.Read();
```

Самый простой способ повторного создания этих ключевых строк в каждом будущем консольном приложении, которые вы будете создавать, описан в следующих разделах.

Сохранение кода на панели элементов

Первый шаг заключается в том, чтобы сохранить эти строки в удобном месте для использования в будущем: в окне панели элементов. При открытом консольном приложении `Program1` в Visual Studio выполните следующие действия.

1. В методе `Main()` класса `Program` выделите три сохраняемые строки (в нашем случае — три ранее упоминавшиеся строки).
2. Убедитесь, что открыто окно панели элементов. (Если это не так, откройте его с помощью пункта меню View⇨Toolbox (Вид⇨Панель элементов).)
3. Перетащите выделенные строки на вкладку General (Общие) окна панели элементов или скопируйте их и поместите на эту панель.

   Панель элементов сохранит эти строки для того, чтобы позже вы могли ими воспользоваться, не набирая их заново. На рис. 1.6 показана панель инструментов с сохраненными строками.

Рис. 1.6. Сохранение части исходного текста на панели инструментов

Повторное использование кода из панели элементов

Теперь, когда у вас есть сохраненный на панели элементов текст шаблона, его можно повторно использовать во всех консольных приложениях, которые вы будете писать. Вот как это сделать.

1. **В Visual Studio создайте новое консольное приложение, как было описано выше, в разделе "Создание исходной программы".**
2. **Щелкните в редакторе в точке исходного текста `Program.cs`, куда бы вы хотели вставить соответствующий текст.**
3. **Убедитесь, что окно панели элементов открыто.**

   Если это не так, откройте его, как сказано выше, в разделе "Сохранение кода на панели элементов".
4. **На вкладе General (Общие) окна панели элементов найдите сохраненный текст и дважды щелкните на нем.**

   Выбранный текст будет вставлен в точке ввода в окне редактора.

Далее вы можете дописать остальные части своего приложения выше этих строк. Теперь у вас есть готовое консольное приложение. Можете немного поиграть с ним, прежде чем перейти к чтению главы 2, "Работа с переменными".

Глава 2

Работа с переменными

В ЭТОЙ ГЛАВЕ...

- Применение переменных C#
- Объявление переменных разных типов
- Работа с числовыми константами
- Изменение типов и вывод типа компилятором

Наиболее фундаментальной из всех концепций программирования является концепция переменной. Переменная C# подобна небольшому ящику, в котором можно хранить разные вещи (в частности, числа) для последующего применения. Термин *переменная* пришел из мира математики.

К сожалению для программистов, C# накладывает ряд ограничений на переменные — ограничений, с которыми не сталкиваются математики. Однако эти ограничения имеют свои причины. Для C# они облегчают понимание того, что вы подразумеваете под переменной определенного типа, а для вас — поиск ошибок в вашем коде. Из этой главы вы узнаете о том, как объявлять, инициализировать и использовать переменные, а также познакомитесь с некоторыми из фундаментальных типов данных в языке C#.

Объявление переменной

Математики работают с числами так, как не в состоянии работать с ними C#. Они свободно вводят переменные при необходимости представить идею определенным образом и используют алгоритмы — множество процедурных шагов, используемых для решения задачи таким образом, что оно имеет смысл для других математиков, моделирующих реальный мир. Математик вполне может сказать или написать следующее:

```
x = y²+2y+1
Если k = y+1, то x = k²
```

Программист должен быть гораздо педантичнее в использовании терминологии. Например, программист на C# может написать следующий код:

```
int n;
n = 1;
```

Первая его строка означает "Выделим небольшое количество памяти компьютера и назначим ему имя `n`". Этот шаг аналогичен, например, абонированию почтового ящика в почтовом отделении и наклейке на него ярлыка. Вторая строка гласит "Сохраним значение 1 в переменной `n`, тем самым заменив им предыдущее хранившееся в ней значение". При использовании аналогии с почтовым ящиком это звучит так: "Откроем ящик, выбросим все, что в нем было, и поместим в него 1".

ЗАПОМНИ!

Знак равенства (=) называется *оператором присваивания*.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Математик говорит: "*n* равно 1". Программист на C# выражается более точно: "Сохраним значение 1 в переменной `n`". Операторы C# указывают компьютеру, что именно вы хотите сделать. Другими словами, операторы — это глаголы, а не существительные. Оператор присваивания берет значение справа от него и сохраняет его в переменной, указанной слева от него. Более подробно об операторах рассказывается в главе 4, "Операторы".

Что такое `int`

В C# каждая переменная имеет фиксированный тип. Абонируя почтовый ящик, вы выбираете ящик интересующего вас размера. Если вы выбрали ящик "для целых чисел", не надейтесь, что в него поместится строка.

В примере из предыдущего раздела вы выбрали ящик, предназначенный для работы с целыми числами; C# называет их `int`. Целые числа — это числа, применяемые для перечисления (1, 2, 3 и т.д.), а также 0 и отрицательные числа (–1, –2, –3 и т.д.).


ЗАПОМНИ!

Перед тем как использовать переменную, ее надо *объявить*. Объявив переменную как `int`, в нее можно помещать целые значения и извлекать их из нее, что продемонстрировано в следующем примере:

```
// Объявляем переменную n
int n;
// Объявляем переменную m и инициализируем ее значением 2
int m = 2;
// Присваиваем значение, хранящееся в m, переменной n
n = m;
```

Первая строка после комментария является *объявлением*, которое создает небольшую область в памяти с именем `n`, предназначенную для хранения целых значений. Начальное значение `n` не определено до тех пор, пока этой переменной не *присвоено* некоторое значение. Второе объявление не только объявляет переменную `m` типа `int`, но и *инициализирует* ее значением 2.


ЗАПОМНИ!

Термин *инициализировать* означает присвоить начальное значение. Инициализация переменной заключается в первом присваивании ей некоторого значения. Вы ничего не можете сказать о значении переменной до тех пор, пока она не будет инициализирована.

Последняя инструкция присваивает значение, хранящееся в `m` (равное 2), переменной `n`. Переменная `n` будет хранить значение 2, пока ей не будет присвоено новое значение (в частности, она не потеряет свое значение при присваивании его переменной `m`).

Правила объявления переменных

Вы можете выполнить инициализацию переменной как часть ее объявления:

```
// Объявление переменной типа int с присваиванием ей
// начального значения 1
int p = 1;
```

Это эквивалентно помещению 1 в ящик `int` в момент его аренды, в отличие от его вскрытия и помещения в него 1 позже.


СОВЕТ

Инициализируйте переменные при их объявлении. Во многих (но не во всех) случаях C# инициализирует переменные вместо вас, но рассчитывать на это нельзя. Например, C# помещает 0 в неинициализированную переменную типа `int`, но компилятор все равно будет

выводить сообщение об ошибке, если вы попытаетесь использовать переменную до ее инициализации. В программе вы можете объявлять переменные где угодно (ну, почти где угодно).


ВНИМАНИЕ!

Однако вы не можете использовать переменную до того, как она будет объявлена, и присваивать ей какие-либо значения. Так, следующие два присваивания *некорректны*:

```
// Это присваивание неверно, поскольку переменной m не
// присвоено значение перед ее использованием
int n;
int m;
n = m;
// Следующее присваивание некорректно в силу того, что
// переменная p не была объявлена до ее использования
p = 2;
int p;
```

И последнее: нельзя дважды объявить одну и ту же переменную в одной области видимости (например, функции).

Вариации на тему `int`

Большинство простых переменных имеют тип `int`. Однако C# позволяет настраивать целый тип для конкретных случаев.

Все целочисленные типы переменных ограничены хранением только целых чисел, но диапазоны этих чисел различны. Например, переменная типа `int` может хранить только целые числа из диапазона примерно от –2 до 2 миллиардов.

Два миллиарда сантиметров — это больше, чем диаметр Земли. Но если этой величины вам не хватает, C# предоставляет еще один целочисленный тип, называемый `long` (сокращение от `long int`), который может хранить гораздо бо́льшие числа за счет увеличения размера "ящика": он занимает 8 байт (64 бит) в отличие от 4-битового `int`.

В C# имеются и другие целочисленные типы, показанные в табл. 2.1.

Таблица 2.1. Размер и диапазон целочисленных типов C#

Тип	Размер, байт	Диапазон значений	Пример использования
`sbyte`	1	От –128 до 127	`sbyte sb = 12;`
`byte`	1	От 0 до 255	`byte b = 12;`
`short`	2	От –32768 до 32767	`short sn = 12345;`

Окончание табл. 2.1

Тип	Размер, байт	Диапазон значений	Пример использования
`ushort`	2	От 0 до 65535	`ushort usn = 62345;`
`int`	4	От –2147483648 до 2147483647	`int n = 1234567890;`
`uint`	4	От 0 до 4294967295	`uint un = 3234567890U;`
`long`	8	От –9223372036854775808 до 9223372036854775807	`long l = 123456789012L;`
`ulong`	8	От 0 до 18446744073709551615	`ulong ul = 123456789012L;`

Как будет рассказано позже, фиксированные значения, такие как 1, тоже имеют тип. По умолчанию считается, что простая константа наподобие 1 имеет тип `int`. Целочисленные константы, отличные от `int`, должны явно указывать свой тип. Так, например, `123U` (обратите внимание на `U`) — это константа типа `uint`, беззнакового целого.

Большинство целых значений — *знаковые* (signed), т.е. они могут представлять наряду с положительными и отрицательные значения. Беззнаковые (unsigned) целые числа могут представлять только неотрицательные значения, но зато их диапазон представления удваивается по сравнению с соответствующими знаковыми типами. Как видно из табл. 2.1, имена большинства беззнаковых типов образуются из знаковых путем добавления префикса `u`.

СОВЕТ

В этой книге беззнаковые целые нам не понадобятся.

Представление дробных чисел

Для множества вычислений необходимы дробные числа, которые никак не могут быть точно представлены целыми числами. Общее уравнение для преобразования температуры в градусах Фаренгейта в температуру в градусах Цельсия демонстрирует это:

```
// Преобразование температуры 41°F
int fahr = 41;
int celsius = (fahr - 32)*(5/9);
```

Это уравнение корректно работает для некоторых значений, например 41 градус по Фаренгейту точно равен 5 градусам по Цельсию.

Попробуем теперь другое значение, например 100°F. Приступим к вычислениям: 100–32=68; 68 умножить на 5/9 дает при использовании целых чисел 37. Но правильный ответ — 37,78; и даже это не совсем верно, так как в действительности правильный ответ — 37,777..., где 7 повторяется до бесконечности.


ЗАПОМНИ!

Тип `int` может представлять только целые числа. Целый эквивалент числа 37,78 — 37. При этом, для того чтобы разместить число в целой переменной, дробная часть числа отбрасывается. Такое действие называется *усечением* (truncation).


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Усечение — это не то же самое, что *округление* (rounding). При усечении отбрасывается дробная часть, а при округлении получается ближайшее целое значение. Так, усечение 1,9 даст 1, а округление — 2.

Для температур значения 37 может оказаться вполне достаточно. Вряд ли ваша одежда при 37,78°C будет существенно отличаться от одежды при 37°C. Но для множества, если не большинства, приложений такое усечение неприемлемо.

На самом деле все гораздо хуже. Тип `int` не в состоянии хранить значение 5/9 и преобразует его в 0. Соответственно, данная формула будет давать нулевое значение `celsius` для любого значения `fahr`. Поэтому даже такой непритязательный человек, как я, сочтет это неприемлемым.

Работа с числами с плавающей точкой

Ограничения, накладываемые на переменные типа `int`, для многих приложений неприемлемы. Обычно главным препятствием является не диапазон возможных значений (двух квинтиллионов 64-битового `long` хватает, пожалуй, для подавляющего большинства задач), а невозможность представления дробных чисел.

В некоторых ситуациях нужны числа, которые могут иметь ненулевую дробную часть и которые математики называют *действительными числами* (real numbers). Всегда находятся люди, удивляющиеся такому названию: неужели целые числа — недействительные?


ЗАПОМНИ!

Обратите внимание на то, что действительное число *может* иметь ненулевую дробную часть, т.е. число 1,5 является действительным так же, как и число 1,0, например 1,0+0,1=1,1. Просто при чтении оставшейся части этой главы не забывайте о наличии запятой (вместо которой в записи чисел в языке программирования C#

используется точка, поэтому в дальнейшем мы будем говорить именно о десятичной точке, а не запятой).

К счастью, C# прекрасно понимает, что такое действительные числа. Они могут быть с плавающей точкой и с так называемым десятичным представлением. Гораздо более распространена плавающая точка.

Объявление переменной с плавающей точкой

Переменная с плавающей точкой может быть объявлена так, как показано в следующем примере:

```
float f = 1.0;
```

После того как вы объявите переменную как `float`, она останется таковой при всех естественных для нее операциях.

В табл. 2.2 рассматриваются использующиеся в C# типы с плавающей точкой. Все переменные этих типов — знаковые (т.е. не существует такой вещи, как беззнаковая переменная с плавающей точкой, не способная представлять отрицательные значения).

Таблица 2.2. Размеры и диапазоны представления типов переменных с плавающей точкой

Тип	Размер, байт	Диапазон значений	Точность, цифр	Пример использования
`float`	4	От 1.5×10^{-45} до 3.4×10^{38}	6–7	`float f = -1.2F;`
`double`	8	От 5.0×10^{-324} до 1.7×10^{308}	15–16	`double d = 1.2;`

ЗАПОМНИ!

Вы можете решить, что тип `float` — это тип по умолчанию для переменных с плавающей точкой, но на самом деле таким типом по умолчанию является `double`. Если вы не определите явно тип для, скажем, `12.3`, C# сделает его `double`.

В столбце “Точность” в табл. 2.2 указано количество значащих цифр, которые может представлять такая переменная. Например, 5/9 на самом деле равно 0,555… с бесконечной последовательностью пятерок. Однако переменная типа `float` имеет точность не более 6 цифр, а это означает, что все цифры после шестой могут быть проигнорированы. Таким образом, 5/9, будучи выраженным в виде `float`, может выглядеть как

```
0.5555551457382
```

Не забывайте, что все цифры после шестой пятерки ненадежны.

То же число 5/9 при использовании типа `double` может выглядеть следующим образом:

```
0.55555555555555557823
```

Тип `double` имеет 15–16 значащих цифр.


СОВЕТ

Числа с плавающей точкой в C# по умолчанию имеют точность `double`, так что применяйте везде тип `double`, если только у вас нет веских причин поступить иначе. Вот как выглядит уравнение для преобразования температуры по Фаренгейту в температуру по Цельсию с использованием арифметики с плавающей точкой:

```
double celsius = (fahr - 32.0) * (5.0 / 9.0)
```

Ограничения переменных с плавающей точкой

Вы можете решить использовать переменные с плавающей точкой везде и всегда, раз уж они так хорошо решают проблему усечения. Да, конечно, они используют немного больше памяти, но ведь сегодня это не проблема? Но дело в том, что у чисел с плавающей точкой имеется ряд ограничений.

Перечисление

Нельзя использовать числа с плавающей точкой для перечисления. Некоторые вещи нужно просто сосчитать (1, 2, 3 и т.д.). Всем известно, что числа 1.0, 2.0, 3.0 можно применять для подсчета количества точно так же, как и 1, 2, 3, но C# этого не знает. Например, при указанной выше точности чисел с плавающей точкой откуда C# знать, что вы не сказали в *действительности* 1.000001?


ЗАПОМНИ!

Независимо от того, находите ли вы эту аргументацию убедительной, вы не можете использовать числа с плавающей точкой для подсчета количества.

Сравнение чисел

Следует быть очень осторожными при сравнении чисел с плавающей точкой. Например, 12,6 может быть представлено как 12,600001. Большинство людей не волнуют такие мелкие добавки в конце числа, но компьютер понимает их буквально, и для C# 12,600000 и 12,600001 — это разные числа.

Так, сложив 1,1 и 1,1, вы можете получить в качестве результата 2,2 или 2,200001. И спросив "Равно ли значение `doubleVariable` 2,2?", вы можете получить совсем не тот ответ, которого ожидаете. Подобные вопросы нужно переформулировать, например, так: "Отличается ли абсолютное значение разности `doubleVariable` от 2,2 менее чем на 0,000001?"; другими словами, равны ли два значения с некоторой допустимой ошибкой?

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Современные процессоры используют небольшой трюк, который несколько уменьшает указанную неприятность: во время работы они применяют специальный формат, в котором для числа с плавающей точкой выделяется 80 бит (или даже 128 бит на новейших процессорах). При округлении такого числа до 64-битового почти всегда получается результат, который вы ожидаете.

Скорость вычислений

Целые числа всегда быстрее, чем числа с плавающей точкой, поскольку целые числа являются менее сложными. Так же, как вы можете рассчитать стоимость чего-то в целых числах гораздо быстрее, чем при использовании этих надоедливых десятичных запятых, так и процессоры работают быстрее с целыми числами.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Процессоры Intel выполняют целочисленные вычисления с использованием внутренней структуры, именуемой регистрами общего назначения, которые могут работать только с целыми числами. Эти же чрезвычайно быстрые регистры используются и для подсчета. Числа с плавающей точкой требуют специальной области процессора, которая может работать с действительными числами и которая называется арифметическо-логическим устройством, и специальных регистров с плавающей точкой, которые не годятся для подсчета. Такие расчеты отнимают больше времени из-за дополнительной работы, которая требуется для чисел с плавающей точкой.

К сожалению, современные процессоры настолько сложны, что невозможно сказать точно, сколько времени вы сэкономите, используя целые числа. Просто знайте, что применение целых чисел, как правило, быстрее, но на самом деле вы не увидите особой разницы, если выполняете длинный список вычислений.

Ограниченность диапазона

В прошлом переменные с плавающей точкой могли представлять значительно больший диапазон чисел, чем целые. Сейчас диапазон представления целых чисел существенно вырос — стоит вспомнить о типе `long`.

ВНИМАНИЕ!

Даже простой тип `float` способен хранить очень большие числа, но количество значащих цифр у него ограничено примерно шестью. Например, `123456789F` означает то же самое, что и `123456000F`. (Что такое `F` в приведенных записях, вы узнаете немного позже.)

Десятичные числа: комбинация целых чисел и чисел с плавающей точкой

Как уже объяснялось в предыдущих разделах, и целые, и десятичные числа имеют свои недостатки. Переменным с плавающей точкой присущи проблемы, связанные с вопросами округления из-за недостаточной точности представления, а целые переменные не могут представлять числа с дробной частью. В некоторых ситуациях совершенно необходимо иметь возможность получить лучшее из обоих миров, а именно — числа, которые:

- подобно числам с плавающей точкой, могут хранить дроби;
- подобно целым числам, должны представлять точные результаты вычислений, т.е. 12,5 должно быть равно 12,5 и ни в коем случае не 12,500001.

К счастью, в C# есть такой тип чисел, называющийся `decimal`. Переменная типа `decimal` может представлять числа от 10^{-28} до 10^{28} — вполне достаточный диапазон значений! И все это делается без проблем, связанных с округлением.

Объявление переменных типа `decimal`

Переменные типа `decimal` объявляются и используются так же, как и переменные других типов:

```
decimal m1 = 100;          // Хорошо
decimal m2 = 100M;         // Еще лучше
```

Объявление `m1` выделяет память для переменной `m1` и инициализирует ее значением 100. В этой ситуации неприятным моментом оказывается то, что 100 имеет тип `int`. Поэтому C# вынужден конвертировать `int` в `decimal` перед инициализацией. К счастью, C# понимает, чего именно вы добиваетесь, и выполняет эту инициализацию для вас.

Лучше всего использовать такое объявление, как объявление переменной `m2` с константой `100M` типа `decimal`. Буква `M` в конце числа указывает, что данная константа имеет тип `decimal`, так что никакого преобразования не требуется.

Сравнение десятичных и целых чисел, а также чисел с плавающей точкой

Создается впечатление, что числа типа `decimal` имеют лишь одни достоинства и лишены недостатков, присущих типам `int` и `double`. Переменные этого типа обладают широким диапазоном представления и не имеют проблем округления.

Однако с числами decimal связаны два значительных ограничения. Поскольку они имеют дробную часть, они не могут использоваться в качестве счетчиков, например, в циклах, о которых пойдет речь в главе 5, "Управление потоком выполнения".

Вторая проблема не менее серьезна и заключается в том, что вычисления с этим типом чисел выполняются гораздо медленнее, чем с простыми целыми числами или даже с числами с плавающей точкой. В простом тесте с 300 000 000 сложений и вычитаний работа с числами decimal оказалась примерно в 50 раз медленнее работы с числами int. Это отношение становится еще хуже для более сложных операций. Кроме того, большинство математических функций, таких как синус или возведение в степень, не имеют версий для работы с числами decimal.

Понятно, что числа типа decimal наиболее подходят для финансовых приложений, в которых исключительно важна точность, но само количество вычислений относительно невелико.

Логичен ли логический тип

И наконец, поговорим о переменных логического типа. Тип bool имеет только два значения — true и false. Это не шутка — целый тип переменных придуман для работы только с двумя значениями.

ВНИМАНИЕ!

Когда-то программисты на C и C++ использовали нулевое значение переменной типа int для обозначения false и ненулевое — для обозначения true. В C# этот фокус не проходит.

Переменная типа bool объявляется следующим образом:

```
bool thisIsABool = true;
```

Не существует никаких способов преобразования переменных bool в другой тип переменных (даже если бы вы могли это делать, это не имело бы никакого смысла). В частности, вы не можете преобразовать bool в int (чтобы, скажем, false превратилось в 0) или в string (чтобы false стало "false").

Символьные типы

Программа, которая выполняет только вычисления, могла бы устроить разве что математиков, страховых агентов и военных (да-да — первые вычислительные машины были созданы для расчета таблиц артиллерийских стрельб).

Однако в большинстве приложений программы должны работать не только с цифрами, но и с буквами.

Язык C# рассматривает буквы как отдельные символы типа `char` и как строки символов типа `string`.

Тип char

Переменная типа `char` может хранить только один символ. Символьная константа выглядит, как символ, окруженный парой одинарных кавычек:

```
char c = 'a';
```

Вы можете хранить любой символ из латинского алфавита, кириллицы, арабского, иврита, японских катаканы и хираганы и массы японских, китайских и корейских иероглифов.

Кроме того, тип `char` может использоваться в качестве счетчика, т.е. его можно применять в циклах, о которых вы узнаете из главы 5, "Управление потоком выполнения". Символы не вызывают никаких проблем, связанных с округлением.

ВНИМАНИЕ!

Переменные типа `char` не включают информации о шрифтах, так что в переменной `char` может храниться, например, вполне корректный иероглиф, но при его выводе без использования соответствующего шрифта вы увидите на экране только мусор.

Специальные символы

Некоторые символы являются непечатными — в том смысле, что они не видны при выводе на экран или принтер. Наиболее очевидным примером такого символа является пробел `' '` (кавычка, пробел, кавычка). Другие символы не имеют буквенного эквивалента, например символ табуляции. Для указания таких символов C# использует обратную косую черту, как показано в табл. 2.3.

Таблица 2.3. Специальные символы

Символьная константа	Значение
`'\n'`	Новая строка
`'\t'`	Табуляция
`'\0'`	Нулевой символ
`'\r'`	Возврат каретки
`'\\'`	Обратная косая черта

Тип string

Еще одним распространенным типом переменных является `string`. В приведенных ниже примерах показано, как объявляются и инициализируются переменные этого типа:

```
// Объявление с отложенной инициализацией
string someString1;
someString1 = "Это строка";

// Инициализация при объявлении предпочтительнее
string someString2 = "Это строка";
```

Константа типа `string`, именуемая также *строковым литералом*, представляет собой набор символов, заключенный в двойные кавычки. Символы в строке могут включать специальные символы, показанные в табл. 2.3. Строка не может быть перенесена на новую строку в исходном тексте на C#, но может содержать символ новой строки, как показано в следующем примере:

```
// Неверная запись строки
string someString = "Это строка
и это строка";
// А вот так - верно
string someString = "Это строка\nи это строка";
```

При выводе на экран при помощи вызова `Console.WriteLine` вы увидите текст, размещенный в двух строках:

```
Это строка
и это строка
```

Строка не является ни перечислимым типом, ни типом-значением — в процессоре не существует встроенного типа строки. Процессор компьютера понимает только числа, но не буквы. Латинская буква *A* в действительности представляет собой для процессора число 65. К строкам применим только один распространенный оператор — оператор сложения, который просто объединяет две строки в одну, например:

```
string s = "Это предложение." + " И это тоже.";
```

Приведенный код присваивает строке `s` значение:

```
"Это предложение. И это тоже."
```

ВНИМАНИЕ!

Строка без символов, записанная как `""` (пара двойных кавычек), является корректной для типа `string` и называется *пустой строкой*. Пустая строка отличается от нулевого символа `'\0'` и от строки, содержащей любое количество пробелов (`" "`).

СОВЕТ

Предпочтительно инициализировать строки значением `String.Empty`, которое означает то же, что и `""`, но его труднее понять неправильно:

```
string mySecretName = String.Empty; // Свойство типа String
```

Кстати, все остальные типы данных в этой главе — *типы-значения* (value types). Строковый тип (о котором более подробно будет рассказано в главе 3, "Работа со строками") типом-значением не является.

Что такое тип-значение

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Типы переменных, описанные в этой главе, имеют фиксированный размер, за исключением типа `string`. Переменные типов с постоянным размером всегда занимают одно и то же количество памяти. Так, при присваивании `a = b` C# может переместить значение `b` в `a`, не предпринимая дополнительных мер, необходимых при работе с типами переменного размера. Кроме того, эти виды переменных хранятся в специальном месте под названием *стек* как фактические значения. Вам не нужно беспокоиться о стеке; просто нужно знать, что он существует как местоположение в памяти. Эта характеристика поясняет, почему данные типы называются *типами-значениями* (value types).

ЗАПОМНИ!

Типы `int`, `double`, `bool` и их "близкие родственники" наподобие беззнакового `int` являются встроенными (в процессор) типами. Встроенные типы переменных и тип `decimal` известны также как типы-значения. Тип `string` не относится ни к тем, ни к другим, поскольку переменная хранится в виде некоторого "указателя" на данные, который называется *ссылкой* (reference). Данные такой строки на самом деле располагаются в другом месте. Можно представить ссылочный тип как адрес дома. Знание адреса дает вам информацию о том, где находится дом, но чтобы воочию его увидеть, нужно пойти по данному адресу.

Типы, о которых речь пойдет в главе 8, "Обобщенность", определяемые программистом и известные как ссылочные типы, не являются ни встроенными, ни типами-значениями. Тип `string` является ссылочным, хотя компилятор C# рассматривает его специальным образом в силу его широкой распространенности.

Сравнение string и char

Хотя строки состоят из символов, тип `string` существенно отличается от типа `char`. Понятно, что имеются некоторые тривиальные отличия. Так, символ заключается в одинарные кавычки, а строка — в двойные. Кроме того, тип `char` — это всегда один символ, так что следующий код не имеет смысла ни в плане сложения, ни в плане конкатенации:

```
char c1 = 'a';
char c2 = 'b';
char c3 = c1 + c2
```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

На самом деле этот код почти компилируем, но его смысл существенно отличается от того, который мы ему приписываем. Язык C# преобразует `c1` и `c2` в значения типа `int`, представляющие собой числовые значения соответствующих символов, после чего складывает полученные значения. Ошибка возникает при попытке сохранить полученный результат в `c3`, так как при размещении значения типа `int` в переменной меньшего размера `char` данные могут быть потеряны. В любом случае эта операция не имеет смысла.

С другой стороны, строка может быть любой длины. Таким образом, конкатенация двух строк вполне осмысленна:

```
string s1 = "a";
string s2 = "b";
string s3 = s1 + s2; // Результат – "ab"
```

В качестве части своей библиотеки C# определяет целый ряд строковых операций, которые будут описаны в главе 3, “Работа со строками”.

СОВЕТ

СОГЛАШЕНИЯ ОБ ИМЕНОВАНИИ

Программирование — и так достаточно сложное дело, чтобы усложнять его еще больше. Чтобы код на C# было легче читать, обычно используются определенные соглашения об именовании переменных, которым желательно следовать, чтобы код был понятен другим программистам.

- **Имена всех объектов, кроме переменных, начинаются с прописной буквы, а имена переменных — со строчной.** Делайте эти имена как можно более информативными (зачастую это приводит к тому, что имена состоят из нескольких слов). Слова для объектов, не являющихся переменными, должны начинаться с прописной буквы, и лучше, если между ними не будет символов подчеркивания, например `ThisIsALongName`.

- **Имена переменных отличаются только тем, что первая буква — строчная: `thisIsALongVariableName`.**

До эры .NET использовалось соглашение, в соответствии с которым первая буква имени переменной указывала ее тип (так называемая "венгерская запись"). Большинство таких букв тривиальны: `f` — для `float`, `d` — для `double`, `s` — для `string` и т.д. Исключением из правила является `n` для `int`. Есть еще одно исключение: по традиции, уходящей в программирование на Фортране, буквы `i`, `j` и `k` также используются как распространенные имена переменных типа `int`.

Венгерская запись постепенно выходит из моды, по крайней мере в кругах программистов .NET. Тем не менее я все еще остаюсь ее поклонником, поскольку она позволяет мне знать тип каждой переменной в программе, не обращаясь к ее объявлению. В последних версиях Visual Studio вы можете просто подвести курсор к переменной и получить информацию о ее типе в окне подсказки, что делает венгерскую запись менее полезной. Однако вместо того, чтобы встревать в "религиозные войны" по поводу того или иного способа именования, выберите тот, который вам по душе, и следуйте ему.

Вычисление високосных лет: `DateTime`

Что если вам нужна программа для выяснения, является ли некоторый год високосным?

Вот алгоритм для поставленной задачи:

```
Год високосный, если он делится на 4,
    но если при этом он делится на 100, то он
    високосный, только если делится на 400.
```

Пока что вы не знаете, как перевести это на язык C#. Но можно просто спросить тип `DateTime` о том, високосный ли некоторый год (тип `DateTime` является типом-значением наподобие `int`):

```
DateTime thisYear = new DateTime(2011, 1, 1);
bool isLeapYear = DateTime.IsLeapYear(thisYear.Year);
```

Результат для 2016 года — `true`, для 2017 — `false`. (Пока что не обращайте внимания на первую строку кода.)

Тип `DateTime` позволяет выполнять около 80 разных операций, например получение названия месяца и дня недели; добавление дней, часов, минут и так далее к конкретной дате; получение количества дней в месяце; вычитание двух дат.

В приведенном далее примере свойство Now типа DateTime используется для установки текущего времени и даты, а один из множества методов DateTime используется для преобразования одного времени в другое:

```
DateTime thisMoment    = DateTime.Now;
DateTime anHourFromNow = thisMoment.AddHours(1);
```

Можно также выделить определенные части типа DateTime:

```
int year = DateTime.Now.Year; // Например, 2008
DayOfWeek dayOfWeek =
    DateTime.Now.DayOfWeek;   // Например, воскресенье
```

Можно выполнить множество других полезных манипуляций типом DateTime:

```
DateTime date     = DateTime.Today;       // Получение даты
TimeSpan time     = thisMoment.TimeOfDay; // Получение времени
TimeSpan duration =
    new TimeSpan(3, 0, 0, 0);             // Продолжительность в днях
DateTime threeDaysFromNow = thisMoment.Add(duration);
```

Первые две строки выделяют интересующую нас информацию из DateTime. Следующие две — добавляют *продолжительность* к DateTime. Продолжительность, или количество времени, отличается от момента времени; вы указываете ее при помощи класса TimeSpan, а моменты времени — при помощи DateTime. Третья строка устанавливает продолжительность TimeSpan равной трем дням, нулю часов, нулю минут и нулю секунд. Четвертая строка добавляет эту продолжительность к объекту DateTime, представляющему текущий момент времени, и дает новый объект DateTime, представляющий время через три дня после текущего.

Вычитание TimeSpan из DateTime из прибавление его к DateTime дает DateTime:

```
TimeSpan duration1 = new TimeSpan(1, 0, 0); // Один час
// Если Today дает 00:00:00, приведенный код даст 01:00:00:

DateTime anHourAfterMidnight =
    DateTime.Today.Add(duration1);
Console.WriteLine("Час после полуночи - {0}",
                  anHourAfterMidnight);

DateTime midnight =
    anHourAfterMidnight.Subtract(duration1);
Console.WriteLine("За час до 01:00 - {0}", midnight);
```

Первая строка создает продолжительность в 1 час, вторая получает время и добавляет к нему этот час. В последующих строках этот час вновь вычитается.

Объявление числовых констант

В жизни очень мало абсолюта, но он присутствует в C#: любое выражение имеет значение и тип. В объявлении наподобие `int n` легко увидеть, что переменная `n` имеет тип `int`. Разумно предположить, что тип результата вычисления `n+1` — также `int`. Но что можно сказать о типе константы `1`?

Тип константы зависит от ее значения и наличия необязательной буквы в конце. Любое целое число величиной примерно до 2 миллиардов (см. табл. 2.1) рассматривается как `int`. Числа, превышающие это значение, трактуются как `long`. Любые числа с плавающей точкой рассматриваются как `double`.

В табл. 2.4 показаны константы, объявленные как имеющие конкретные типы, т.е., в частности, с буквенными дескрипторами в конце. Строчные эти буквы или прописные — значения не имеет, например записи `1u` и `1U` равноценны.

Таблица 2.4. Объявление констант с их типом

Константа	Тип
`1`	`int`
`1U`	`unsigned int`
`1L`	`long int` (избегайте использования строчной `l` — она слишком похожа на единицу, `1`)
`1.0`	`double`
`1.0F`	`float`
`1M`	`decimal`
`true`	`bool`
`false`	`bool`
`'a'`	`char`
`'\n'`	`char` (символ новой строки)
`'\x123'`	`char` (символ с шестнадцатеричным числовым значением 123)
`"a string"`	`string`
`""`	`string` (пустая строка)

Преобразование типов

Человек не рассматривает числа, используемые для счета, как разнотипные. Например, нормальный человек (не программист на C#) не станет задумываться, глядя на число 1, знаковое оно или беззнаковое, "короткое" или "длинное". Хотя для C# все эти типы различны, даже он понимает, что все они тесно связаны между собой. Например, в приведенном далее фрагменте исходного текста величина типа `int` преобразуется в `long`:

```
int intValue = 10;
long longValue;
longValue    = intValue;    // Это присваивание корректно
```

Переменная типа `int` может быть преобразована в `long`, поскольку любое значение типа `int` может храниться в переменной типа `long` и оба типа представляют собой числа, пригодные для перечислений. C# выполняет это преобразование автоматически, без каких-либо комментариев. Такое преобразование типов называется *неявным* (implicit).

Однако преобразование в обратном направлении может вызвать проблемы. Например, приведенный далее фрагмент исходного текста содержит ошибку:

```
long longValue = 10;
int intValue;
intValue       = longValue;    // Неверно!
```

СОВЕТ

Некоторые значения, которые могут храниться в переменной `long`, не помещаются в переменной типа `int` (ну, например, 4 миллиарда). C# в такой ситуации генерирует сообщение об ошибке, поскольку в процессе преобразования данные могут быть утеряны. Ошибку такого рода обычно довольно сложно обнаружить.

Но что если вы точно знаете, что такое преобразование вполне допустимо? Например, несмотря на то что переменная `longValue` имеет тип `long`, в данной конкретной программе ее значение не может превышать `100`. В этом случае преобразование переменной `longValue` типа `long` в переменную `intValue` типа `int` совершенно корректно.

Вы можете пояснить C#, что отлично понимаете, что делаете, посредством *приведения типов*:

```
long longValue = 10;
int intValue;
intValue       = (int)longValue;    // Все в порядке
```

При приведении вы размещаете имя требуемого типа в круглых скобках непосредственно перед преобразуемым значением. Приведенная выше запись

гласит: "Не волнуйся и преобразуй `longValue` в тип `int` — я знаю, что делаю, и всю ответственность беру на себя". Конечно, такое утверждение в ретроспективе может оказаться излишне самоуверенным, но зачастую оно совершенно справедливо.

Перечислимые числа могут быть преобразованы в числа с плавающей точкой автоматически, но для обратного преобразования необходим оператор приведения типов, например:

```
double doubleValue = 10.0;
long   longValue   = (long)doubleValue;
```

Все приведения к типу `decimal` и из него нуждаются в операторе приведения типов. В действительности все числовые типы могут быть преобразованы в другие числовые типы с помощью такого оператора. Однако ни `bool`, ни `string` не могут быть непосредственно приведены ни к какому иному типу.

Встроенные методы C# могут преобразовывать числа, символы или логические переменные в их строковые "эквиваленты". Например, вы можете преобразовать значение `true` типа `bool` в строку `"true"`; однако такое преобразование нельзя рассматривать как непосредственное. Эти два значения — совершенно разные вещи.

Позвольте компилятору C# вывести типы данных

Пока что везде в этой книге — ну ладно, в этой главе, — объявляя переменную, мы *всегда* указывали ее точный тип:

```
int i = 5;
string s = "Hello C#";
double d = 1.0;
```

Но можно переложить часть работы на плечи компилятора C#, воспользовавшись ключевым словом `var`:

```
var i = 5;
var s = "Hello C# 4.0";
var d = 1.0;
```

В этом случае компилятор сам *выводит* тип данных — он смотрит на то, что находится в правой части присваивания, чтобы выяснить, какой тип требуется в левой части.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В главе 3, "Работа со строками", рассматривается, как вычисляется тип выражений наподобие приведенных выше. Но такая информация вам, скорее всего, не понадобится — компилятор со всем справится и без вас. Предположим, например, что есть инициализирующее выражение наподобие

```
var x = 3.0 + 2 - 1.5;
```

Компилятор в состоянии вывести, что `x` — значение типа `double`. Он видит `3.0` и `1.5` и знает, что они имеют тип `double`. Затем он видит, что `2` имеет тип `int`, который компилятор может *неявно* конвертировать в `double` для выполнения вычислений. В результате все члены выражения инициализации переменной `x` имеют тип `double`, так что *выведенный тип* `x` также представляет собой `double`.

Так что теперь для объявления переменной достаточно ключевого слова `var` и инициализирующего выражения, остальное компилятор сделает сам:

```
var aVariable = <выражение инициализации>;
```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Если вы работали с такими языками сценариев, как JavaScript или VBScript, то вы должны были использовать тип данных "все в одном". В VBScript такой тип данных называется `Variant` — тип данных, который может быть чем угодно. Не является ли `var` в C# обозначением типа данных `Variant`? Ни в коем случае. Объект, объявленный с применением ключевого слова `var`, *имеет определенный тип данных* C#, такой как `int`, `string` или `double`. Вы просто не объявляете его.

Чем же в действительности являются переменные, объявленные как `var`? Давайте взглянем на следующий фрагмент кода:

```
var aString = "Hello C# 3.0";
Console.WriteLine(aString.GetType().ToString());
```

Здесь инструкция `WriteLine` вызывает метод `String.GetType()` объекта `aString`, чтобы получить его тип данных C#. Затем для полученного объекта вызывается метод `ToString()`, позволяющий вывести название типа. Вот почему в окне консоли вы увидите

```
System.String
```

Это доказывает корректный вывод типа переменной `aString`.

СОВЕТ

В большинстве случаев не стоит использовать ключевое слово `var`. Оставьте его на те случаи, когда это необходимо. Явное указание типа переменной делает исходный текст понятнее любому, кто будет его читать.

Ниже будут приведены примеры, в которых применение `var` совершенно необходимо, так что я буду использовать его во многих местах книги — часто даже там, где без него, строго говоря, можно было бы обойтись. Посмотрев на примеры, вы сами решите, как и когда применять это ключевое слово.


СОВЕТ

Вы можете встретить различные применения ключевого слова `var`, например, с массивами или коллекциями данных (глава 6, "Глава для коллекционеров") или с анонимными типами (часть 2, "Объектно-ориентированное программирование на C#").

Начиная с C# 4.0 типы становятся еще более гибкими, чем `var`: тип `dynamic` делает по сравнению с `var` шаг вперед.

Тип `var` заставляет компилятор вывести тип переменной на основе ожидаемых вводимых данных. Ключевое слово `dynamic` делает это во время выполнения, используя инструментарий под названием "Dynamic Language Runtime". Дополнительную информацию о `dynamic` вы найдете в части 3, "Вопросы проектирования на C#".

Глава 3

Работа со строками

В ЭТОЙ ГЛАВЕ...

- Основные операции со строками в C#
- Сравнение, обрезание, разделение, конкатенация строк
- Анализ считанной строки
- Форматирование выводимых строк

Во многих приложениях тип `string` можно рассматривать как один из встроенных типов-значений наподобие `int` или `char`. К строкам применимы некоторые из операций, зарезервированных для встроенных типов, например:

```
int i = 1;         // Объявление и инициализация int
string s = "abc"; // Объявление и инициализация string
```

В других отношениях, как видно из приведенного далее фрагмента, строки можно рассматривать как пользовательский класс (о классах речь пойдет в разделе 2, "Объектно-ориентированное программирование на C#"):

```
string s1 = new String();
string s2 = "abcd";
int lengthOfString = s2.Length;
```

Так что же такое `string` — тип-значение или класс? На самом деле `String` — это класс, который в силу его широкой распространенности C# трактует специальным образом. Например, ключевое слово `string` является синонимом имени класса `String`, как видно из следующего фрагмента исходного текста:

```
String s1 = "abcd"; // Присваивание строкового литерала
                    // объекту класса String
string s2 = s1;     // Присваивание объекта класса String
                    // строковой переменной
```

В этом примере переменная `s1` объявлена как объект класса `String` (обратите внимание на прописную `S` в начале имени), в то время как `s2` объявлена как просто `string` (со строчной `s` в начале имени). Однако эти два присваивания демонстрируют, что `string` и `String` — это одинаковые (или совместимые) типы.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В действительности то же самое справедливо и для других встроенных типов. Даже тип `int` имеет соответствующий класс `Int32`, `double` — класс `Double` и т.д.; отличие в том, что `string` и `String` — это действительно одно и то же.

В остальной части главы рассматриваются `String` и `string` и действия, которые можно с ними выполнять.

Неизменяемость строк

Запомните одну до сих пор неизвестную вам вещь: после того как объект `string` создан, изменить его нельзя. Несмотря на то, что можно говорить о модификации строки, в C# нет операции, модифицирующей реальный объект `string`. Внешне создается впечатление, что множество операторов модифицируют объекты `string`, с которыми работают, но это не так — они всегда возвращают модифицированную строку как новый объект `string`.

Например, операция `"Его имя - "+"Randy"` не изменяет ни одну из объединяемых строк, а генерирует новую строку `"Его имя - Randy"`. Одним из побочных эффектов такого поведения является то, что вы не должны беспокоиться о том, что кто-то изменит строку без вашего ведома. Рассмотрим следующую программу:

```
// ModifyString - методы класса String не модифицируют сам
// объект (s.ToUpper() не изменяет строку s; вместо этого он
// возвращает новую преобразованную строку)
using System;

namespace ModifyString
{
  class Program
  {
    public static void Main(string[] args)
    {
      // Создание объекта Student
      Student s1 = new Student();
      s1.Name = "Jenny";
```

```
        // Создаем новый объект с тем же именем
        Student s2 = new Student();
        s2.Name = s1.Name;

        // "Изменение" имени объекта s1 не изменяет сам
        // объект, поскольку ToUpper() возвращает новую
        // строку, не влияя на оригинал
        s2.Name = s1.Name.ToUpper();
        Console.WriteLine("s1 - " + s1.Name +
                          ", s2 - " + s2.Name);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
}

// Student - простейший класс, содержащий строку
class Student
{
    public String Name;
}
```

О том, что такое классы, мы поговорим в части 2, "Объектно-ориентированное программирование на C#". Пока же вы можете видеть, что класс `Student` содержит переменную `Name` типа `String`. Объекты класса `Student` `s1` и `s2` созданы так, что их члены `Name` указывают на одни и те же строковые данные. Вызов метода `ToUpper()` преобразует строку `s1.Name`, изменяя все ее символы на прописные. Никаких проблем, связанных с тем, что `s1` и `s2` указывают на один объект, не возникает, поскольку метод `ToUpper()` не изменяет `Name`, а создает новую строку, записанную прописными буквами.

Вот что выводит на экран приведенная выше программа:

```
s1 - Jenny, s2 - JENNY
Нажмите <Enter> для завершения программы...
```

Это свойство строк называется *неизменностью* (или неизменяемостью — immutability).

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Неизменность строк очень важна для строковых констант. Строка наподобие `"Это строка"` представляет собой вид строковой константы, как `1` представляет собой константу типа `int`. Компилятор, таким образом, может заменить все обращения к одинаковым константным строкам обращением к одной константной строке, что снижает размер получающейся программы. Такое поведение компилятора было бы невозможным, если бы строки могли изменяться.

Основные операции над строками

Программисты C# выполняют больше операций над строками, чем пластические хирурги над стареющими актрисами. Практически в каждой программе имеется "сложение" строк, как в следующем примере:

```
string name = "Randy";
Console.WriteLine("Его имя - " + // + означает конкатенацию
                  name);
```

Этот специальный оператор обеспечивается классом `String`. Однако класс `String` предоставляет и другие методы для работы со строками. Их полный список можно найти в разделе "String class" предметного указателя справочной системы. С некоторыми из них вы познакомитесь в этой главе, в том числе со следующими.

- Сравнение строк на равенство или в алфавитном порядке
- Изменение и преобразование строк: замена частей, регистра символов, преобразования между строками и иными типами
- Обращение к отдельным символам строки
- Поиск символов или подстрок в строке
- Обработка информации из командной строки
- Форматирование вывода
- Работа с использованием `StringBuilder`

Сравнение строк

Часто требуется выполнить сравнение двух строк. Например, ввел ли пользователь ожидаемое значение? Возможно, у вас есть список строк и нужно отсортировать их в алфавитном порядке? Лучшая практика призывает избегать стандартных операторов сравнения == и !=, используя встроенные функции сравнения. Дело в том, что могут быть различные нюансы при их работе, так что эти операторы не всегда работают так, как ожидалось. Кроме того, применение функций сравнения делает код яснее и легче в поддержке. Дополнительную информацию по этому вопросу вы можете найти в статье по адресу `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/how-tocompare-strings`.

Проверка равенства: метод Compare()

Множество операций рассматривает строку как единый объект, например метод Compare(), который сравнивает две строки, вводя для них отношение "меньше-больше".

- Если левая строка *больше* правой, Compare(left,right) возвращает 1.
- Если левая строка *меньше* правой, Compare(left,right) возвращает -1.
- Если строки равны, Compare(left,right) возвращает 0.

Вот как выглядит алгоритм работы Compare(), записанный с использованием *псевдокода*:

```
compare(string s1, string s2)
{
  // Циклический проход по всем символам строк, пока один из
  // символов одной строки не окажется больше
  // соответствующего ему символа второй строки
  foreach (для каждого) символа более короткой строки
    if (числовое значение символа строки s1 > числового
        значения символа строки s2)
      return 1
    if (числовое значение символа строки s1 < числового
        значения символа строки s2)
      return -1
  // Если все символы совпали, но строка s1 длиннее строки
  // s2, то она считается больше строки s2
  if В строке s1 остались символы
    return 1
  // Если все символы совпали, но строка s2 длиннее строки
  // s1, то она считается больше строки s1
  if В строке s2 остались символы
    return -1
  // Если все символы строк попарно одинаковы и строки
  // имеют одну и ту же длину, то это одинаковые строки
  return 0
}
```

Таким образом, "abcd" больше "abbd", а "abcde" больше "abcd". Как правило, в реальных ситуациях важно не то, какая из строк больше, а равны ли две строки одна другой. Какая строка больше/меньше — важно в случае сортировки строк.


ЗАПОМНИ!

Метод Compare() возвращает значение 0, если две строки идентичны. В приведенной далее тестовой программе это значение используется для выполнения ряда операций, когда программа встречает

некоторую строку или строки. BuildASentence запрашивает у пользователя ввод строк текста. Эти строки объединяются с предыдущими для построения единого предложения. Программа завершает работу, если пользователь вводит слово EXIT, exit, QUIT или quit.

```
// BuildASentence - данная программа конструирует
// предложение путем конкатенации пользовательского ввода
// до тех пор, пока пользователь не введет команду
// завершения. Эта программа демонстрирует использование
// проверки равенства строк
using System;
namespace BuildASentence
{
  public class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Каждая введенная вами строка "
                        "будет добавляться в предложение, "
                        "пока вы не введете EXIT или QUIT");
      // Запрашиваем пользовательский ввод и соединяем
      // вводимые пользователем фразы в единое целое, пока
      // не будет введена команда завершения работы
      string sentence = "";
      for(;;)
      {
        // Получение очередной строки
        Console.WriteLine("Введите строку");
        string line = Console.ReadLine();
        // Выход при вводе команды завершения

        string[] terms = { "EXIT", "exit", "QUIT", "quit" };
        // Сравниваем введенную строку с командами выхода
        bool quitting = false;
        foreach (string term in terms)
        {
          // Прекращение цикла при совпадении
          if (String.Compare(line, term) == 0)
          {
            quitting = true;
          }
        }
        if (quitting == true)
        {
          break;
        }

        // В противном случае добавление введенного к строке
        sentence = String.Concat(sentence, line);
        // Обратная связь
        Console.WriteLine("\nВы ввели: " + sentence);
      }
```

```
            Console.WriteLine("\nПолучилось:\n" + sentence);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

После краткого описания своих действий программа создает пустую строку для предложения, `sentence`, после чего входит в "бесконечный" цикл.


ЗАПОМНИ!

Конструкции `while(true)` и `for(;;)` представляют собой бесконечные циклы, выход из которых осуществляется посредством оператора `break` (выход из цикла) или `return` (выход из программы). Эти два цикла эквивалентны, и оба встречаются в реальных программах. О циклах подробно рассказывается в главе 5, "Управление потоком выполнения".

Далее программа предлагает пользователю ввести строку текста, которую затем считывает с помощью метода `ReadLine()`. После прочтения строки программа проверяет, не является ли введенная строка командой завершения работы.

Раздел завершения программы определяет массив строк `terms` и переменную типа `bool` с именем `quitting`, получающую при инициализации значение `false`. Каждый элемент в массиве `terms` представляет собой одну из искомых строк. Все эти строки приводят к завершению работы программы.


ВНИМАНИЕ!

В массив включены как строка `"EXIT"`, так и строка `"exit"`, поскольку функция `Compare()` по умолчанию рассматривает их как различные (так что другие варианты написания этого слова, такие как `"eXit"` или `"Exit"`, не будут восприняты программой в качестве команды завершения).

Раздел завершения программы циклически просматривает все элементы массива команд завершения работы и сравнивает их с переданной строкой. Если функция `Compare()` сообщает о соответствии строки одному из образцов команд завершения, переменная `quitting` получает значение `true`; если же до завершения цикла соответствие не найдено, то после выхода из цикла она остается равной `false`. В последнем случае программа продолжает работу, добавляя введенную строку в конец предложения при помощи вызова `String.Concat()`. Программа выводит получившийся результат и продолжает работу. Итерации по массиву — отличный способ проверки на равенство одному из возможных значений. Вот пример вывода программы `BuildASentence`.

```
Каждая введенная вами строка будет добавляться в
предложение, пока вы не введете EXIT или QUIT
Введите строку
Программирование на C#

Вы ввели: Программирование на C#
Введите строку
 - сплошное удовольствие

Вы ввели: Программирование на C# - сплошное удовольствие
Введите строку
EXIT

Получилось:
Программирование на C# - сплошное удовольствие
Нажмите <Enter> для завершения программы...
```

Текст, введенный пользователем, здесь выделен полужирным шрифтом.

Сравнение без учета регистра

Метод `Compare()`, использованный в предыдущем примере, рассматривает строки `"EXIT"` и `"exit"` как различные. Однако имеется вторая версия функции `Compare()`, которой передаются три аргумента. Третий аргумент этой функции указывает, следует ли при сравнении игнорировать регистр букв (значение `true`) или не следует (значение `false`).

Следующая версия раздела завершения программы возвращает значение `true`, какими бы буквами ни была введена команда завершения — прописными, строчными или некоторой их комбинацией:

```
// Проверяет, равна ли переданная строка строкам exit
// или quit, независимо от регистра используемых букв
if (String.Compare("exit", source, true) == 0) ||
   (String.Compare("quit", source, true) == 0)
{
    quitting = true;
}
```

Эта версия проще предыдущей версии с использованием цикла. Ей не надо заботиться о регистре символов, и она может обойтись всего лишь двумя условными выражениями, так как ей достаточно рассмотреть только два варианта команды завершения программы — `QUIT` и `EXIT`.

Изменение регистра

В некоторых случаях нужно перевести все символы строки (или только один) в другой регистр, т.е. строчные сделать прописными или наоборот.

Отличие строк в разных регистрах

Мне не нравится этот способ, но вы можете использовать конструкцию `switch` (см. главу 5, "Управление потоком выполнения") для поиска действий для конкретной строки. Обычно конструкция `switch` применяется для сравнения значения переменной с некоторым набором возможных значений, однако эту конструкцию можно применять и для объектов `string`. Вот как выглядит версия раздела завершения с использованием конструкции `switch`:

```
switch(line)
{
    case "EXIT":
    case "exit":
    case "QUIT":
    case "quit":
        return true;
}
return false;
```

Такой подход работает постольку, поскольку выполняется сравнение только предопределенного ограниченного количества строк. Цикл `for()` представляет собой значительно более гибкий подход, а применение функции `Compare()`, нечувствительной к регистру, существенно повышает возможности программы по "пониманию" введенного пользователем.

Преобразование символов строки в символы верхнего или нижнего регистра

Предположим, у вас имеется строка в нижнем регистре и вам надо преобразовать ее в верхний регистр. Вы можете использовать метод `ToUpper()`:

```
string lowcase = "armadillo";
string upcase  = lowcase.ToUpper(); // ARMADILLO
```

Аналогично строку в верхнем регистре можно преобразовать в строку в нижнем регистре при помощи метода `ToLower()`.

А что если вам надо перевести в верхний регистр только одну первую букву строки? Приведенный ниже код решает эту задачу (хотя в последнем разделе главы вы познакомитесь с лучшим способом):

```
string name = "chuck";
string properName =
    char.ToUpper(name[0]).ToString() +
    name.Substring(1, name.Length - 1);
```

Идея решения заключается в том, чтобы выделить первый символ из строки `name` (т.е. `name[0]`), преобразовать его в односимвольную строку при помощи метода `ToString()`, а затем прибавить к ней часть строки `name`, оставшуюся после удаления первого символа вызовом `Substring()`.

Выяснить, находится ли строка в верхнем или в нижнем регистре, можно при помощи следующей конструкции `if`:

```
if (string.Compare(line.ToUpper(CultureInfo.InvariantCulture),
                   line, false) == 0) ... // Истинно, если строка
                                          // в верхнем регистре
```

Здесь метод `Compare()` сравнивает версию строки `line` в верхнем регистре с исходной строкой `line`. Если строка изначально в верхнем регистре, строки должны совпадать. Что означает "головоломное" выражение `CultureInfo.InvariantCulture`, можно узнать из статьи по адресу `https://msdn.microsoft.com/library/system.globalization.cultureinfo.invariantculture.aspx`. Проверка, находится ли строка в нижнем регистре, выполняется аналогично — нужно только добавить оператор логического отрицания (`!`) перед вызовом `Compare()`. В качестве альтернативы можно использовать цикл, как описано в следующем разделе.

Цикл по строке

Можно обратиться к отдельным символам строки в цикле `foreach`. Приведенный далее код проходит по символам строки и выводит каждый из них на консоль, т.е. просто выводит строку иным способом:

```
string favoriteFood = "cheeseburgers";
foreach(char c in favoriteFood)
{
    Console.Write(c); // Вывод символа
}
Console.WriteLine();
```

Циклом можно воспользоваться и для выяснения, находится ли вся строка в верхнем регистре (см. предыдущий раздел):

```
bool isUppercase = true; // Предполагаем, что строка
                         // в верхнем регистре
foreach(char c in favoriteFood)
{
    if(!char.IsUpper(c))
    {
        isUppercase = false; // Предположение опровергнуто,
        break;               // можно выходить
    }
}
```

В конце цикла переменная `isUppercase` принимает значение либо `true`, либо `false`. Как было показано в последнем примере предыдущего раздела, обращаться к отдельным символам строки можно с использованием записи индекса массива.

ЗАПОМНИ!

Массивы в C# начинаются с нулевого элемента, так что, если вам нужен первый символ, запрашивайте индекс `[0]`. Если нужен третий символ, запрашивайте индекс `[2]`.

```
char thirdChar =
    favoriteFood[2]; // Первое 'e' в "cheeseburgers"
```

Поиск в строках

Что если необходимо найти в строке определенное слово или некоторый символ? Например, вам нужен индекс этой подстроки, чтобы воспользоваться методами `Substring()`, `Replace()`, `Remove()` или некоторыми другими. Из этого раздела вы узнаете, как находить в строке отдельные символы или подстроки. Здесь я буду использовать в качестве примера переменную `favoriteFood` из предыдущего раздела.

Как искать

Простейший способ поиска отдельного символа — при помощи метода `IndexOf()`:

```
int indexOfLetterS = favoriteFood.IndexOf('s'); // 4
```

Класс `String` имеет и другие методы поиска как индивидуальных символов, так и подстрок.

- Метод `IndexOfAny()` получает массив символов и ищет в строке любой из них, возвращая индекс первого найденного символа.

  ```
  char[] charsToLookFor = { 'a', 'b', 'c' };
  int indexOfFirstFound =
      favoriteFood.IndexOfAny(charsToLookFor); // 0
  ```

  Этот вызов можно записать в сокращенном виде:

  ```
  int index = name.IndexOfAny(new char[] { 'a', 'b', 'c' });
  ```

- Метод `LastIndexOf()` находит не первый встреченный символ, а, наоборот, последний.
- Метод `LastIndexOfAny()` работает подобно методу `IndexOfAny()`, но начинает работу с конца строки.
- Метод `Contains()` возвращает `true`, если данная подстрока входит в состав строки :

  ```
  if(favoriteFood.Contains("ee")) ... // true
  ```

- Метод `Substring()` возвращает подстроку, если это возможно, или пустую строку в противном случае:

  ```
  string sub = favoriteFood.Substring(6, favoriteFood.Length - 6);
  ```

  Метод `Substring()` будет более подробно рассмотрен далее в этой главе.

Пуста ли строка

Как выяснить, пуста ли целевая строка (`""`) или имеет значение `null` (т.е. ей не присвоено никакое значение, даже пустая строка)? Для этого можно воспользоваться методом `IsNullOrEmpty()`:

```
bool notThere = string.IsNullOrEmpty(favoriteFood); // false
```

Обратите внимание на то, как вызывается метод `IsNullOrEmpty()`: **`string`**`.IsNullOrEmpty(s)`.

Сделать строку пустой можно двумя способами:

```
string name = "";
string name = string.Empty;
```

Получение введенной пользователем информации

Распространенной подзадачей в консольных приложениях является получение введенной пользователем информации. Считывать информацию необходимо как строку (все, что поступает от пользователя, поступает в виде строки). Затем часто требуется *проанализировать*, или *разобрать* (parse), входную строку, например, чтобы выделить из нее числовые данные.

Удаление пробельных символов

Сначала рассмотрим, как убрать все пробельные символы с обоих концов строки (под термином *пробельный символ* (white space) подразумеваются символы, обычно не отображаемые на экране, например пробел, символ новой строки (`\n`) или табуляции (`\t`); иногда встречается символ возврата каретки (`\r`)). Для этого можно воспользоваться методом `Trim()`:

```
// Удаляем пробельные символы с концов строки
random = random.Trim();
```

Класс `String` предоставляет также методы `TrimFront()` и `TrimEnd()` для удаления пробельных символов с одной стороны строки, и вы можете передать им массивы символов, которые также должны рассматриваться как пробельные. Например, можете убрать расположенный перед денежными значениями символ `'$'`. Такая очистка строк делает их анализ более простым. Все перечисленные методы возвращают новые строки.

Анализ числового ввода

Программа может считывать с клавиатуры по одному символу за раз, но в таком случае вам придется самостоятельно обрабатывать ввод символа новой строки и т.п. Более простой подход состоит в том, чтобы считать строку полностью, а затем разобрать ее на отдельные символы. Посимвольный анализ строки время от времени необходим, но некоторые программисты злоупотребляют этой методикой.

Метод `ReadLine()` используется для считывания объекта типа `string`. Программа, которая ожидает числовой ввод, должна эту строку соответствующим образом преобразовать в числа. C# предоставляет программисту класс `Convert` со всем необходимым для этого инструментарием, в частности методами для преобразования строки в каждый из встроенных числовых типов. Так, следующий фрагмент исходного текста считывает число с клавиатуры и сохраняет его в переменной типа `int`:

```
string s = Console.ReadLine(); // Данные вводятся как строка
int n = Convert.ToInt32(s);    // и преобразуются в число
```

ЗАПОМНИ!

Другие методы для преобразования еще более очевидны: `ToDouble()`, `ToFloat()`, `ToBoolean()`. Метод `ToInt32()` выполняет преобразование в 32-битовое знаковое целое число (вспомните, что 32 бит — это размер обычного `int`), так что эта функция выполняет преобразование строки в число типа `int`; для преобразования строки в число типа `long` используется функция `ToInt64()`.

Метод `Convert()`, встретив "неправильный" символ, может выдать некорректный результат, так что вам следует убедиться, что строка содержит именно те данные, которые ожидаются, и в ней нет никаких "неподходящих" символов.

Приведенная далее функция возвращает значение `true`, если переданная ей строка состоит только из цифр. Такая функция может быть вызвана перед функцией преобразования строки в целое число, поскольку число может состоять только из цифр.

```
// IsAllDigits - возвращает true, если все символы строки
// являются цифрами
public static bool IsAllDigits(string raw)
{
  // Убираем все лишнее с концов строки. Если при этом в
  // строке ничего не остается, значит, строка не
  // представляет собой число
  string s = raw.Trim(); // Игнорируем пробельные символы
  if (s.Length == 0)
  {
    return false;
  }
  // Циклически проходим по всем символам строки
```

```
  for(int index = 0; index < s.Length; index++)
  {
    // Наличие в строке символа, не являющегося цифрой,
    // говорит о том, что это не число
    if (Char.IsDigit(s[index]) == false)
    {
      return false;
    }
  }
  // Все в порядке: строка состоит только из цифр
  return true;
}
```


ЗАПОМНИ!

Вообще-то, для чисел с плавающей точкой в строке может указываться эта самая точка; кроме того, перед числом может находиться минус. Но сейчас интерес представляют не эти частности, а сама идея.

Метод `IsAllDigits` сначала удаляет все ненужные пробельные символы с обоих концов строки. Если после этого строка оказывается пустой, значит, она состояла из пробельных символов и числом не является. Если строка остается не пустой, функция проходит по всем ее символам. Если какой-то из символов оказывается не цифрой, функция возвращает `false`, указывая, что переданная ей строка не является числом. Возврат функцией значения `true` означает, что все символы строки — цифры, так что строка, по всей видимости, представляет собой некоторое числовое значение. Следующая демонстрационная программа считывает вводимое пользователем число и выводит его на экран:

```
// IsAllDigits - демонстрационная программа, иллюстрирующая
// применение функции IsAllDigits
using System;

namespace IsAllDigits
{
  class Program
  {
    public static void Main(string[] args)
    {
      // Ввод строки с клавиатуры
      Console.WriteLine("Введите целое число");
      string s = Console.ReadLine();

      // Проверка, может ли эта строка быть числом
      if (!IsAllDigits(s))
      {
        Console.WriteLine("Это не число!");
      }
      else
      {
        // Преобразование строки в целое число
        int n = Int32.Parse(s);
```

```
        // Выводим число, умноженное на 2
        Console.WriteLine("2 * {0} = {1}", n, 2 * n);
      }
      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
    }
    // IsAllDigits - возвращает true, если все символы строки
    // являются цифрами
    public static bool IsAllDigits(string raw)
    {
      // Тело функции было рассмотрено ранее и здесь оно для
      // краткости опущено
    }
  }
}
```

Программа считывает строку, вводимую пользователем с клавиатуры, после чего проверяет ее с помощью функции `IsAllDigits`. Если функция возвращает `false`, программа выводит предупреждающее сообщение для пользователя. В противном случае программа преобразует строку в число с помощью функции `Int32.Parse()`, которая представляет собой альтернативу `Convert.ToInt32()`. И наконец, программа выводит полученное число и его удвоенное значение (что должно доказывать корректность преобразования строки в число).

```
Введите целое число
1A3
Это не число!
Нажмите <Enter> для завершения программы...
```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Можно просто попытаться использовать функцию `Convert()` для преобразования строки в число и обработать возможные исключения, генерируемые функцией преобразования. Однако, скорее всего, функция не сгенерирует исключения, а вернет некорректный результат; например в приведенном выше примере с вводом в качестве числа `1A3` вернет значение `1`. Вы должны проверять вводимые данные самостоятельно.

СОВЕТ

Можно также воспользоваться методом `Int32.TryParse(s,n)`, который возвращает `false`, если анализ выполнен неудачно, и `true`, если все в порядке. Этот метод преобразует найденное число во второй параметр, переменную типа `int`. Данная функция не генерирует исключений, а пример ее использования приведен в следующем разделе.

Обработка последовательности чисел

Зачастую программы получают в качестве вводимых данных строку, состоящую из нескольких чисел. Воспользовавшись методом `String.Split()`, вы сможете легко разбить строку на несколько подстрок, по одной для каждого числа, и работать с ними по отдельности.

Функция `Split()` преобразует единую строку в массив строк меньшего размера с применением указанного символа-разделителя. Например, если вы скажете функции `Split()`, что следует использовать в качестве разделителя запятую, строка `"1,2,3"` превратится в три строки — `"1"`, `"2"` и `"3"`. (*Символом-разделителем* может быть любой символ, используемый для разделения элементов коллекций.) В приведенной далее демонстрационной программе метод `Split()` применяется для ввода последовательности чисел для суммирования:

```
// ParseSequenceWithSplit - считывает последовательность
// разделенных запятыми чисел, разделяет ее на отдельные
// целые числа и суммирует их
namespace ParseSequenceWithSplit
{
  using System;

  class Program
  {
    public static void Main(string[] args)
    {
      // Приглашение пользователю ввести последовательность
      // целых чисел
      Console.WriteLine("Введите последовательность целых" +
                        " чисел, разделенных запятыми:");

      // Считывание строки текста
      string input = Console.ReadLine();
      Console.WriteLine();

      // Преобразуем строку в отдельные подстроки с
      // использованием в качестве символов-разделителей
      // запятых и пробелов
      char[] dividers = {',', ' '};
      string[] segments = input.Split(dividers);

      // Конвертируем каждую подстроку в число
      int sum = 0;
      foreach(string s in segments)
      {
        // (Пропускаем пустые подстроки)
        if (s.Length > 0)
        {
          // Пропускаем строки, не являющиеся числами
```

```
            if (IsAllDigits(s))
            {
              // Преобразуем строку в 32-битовое целое число
              int num = 0;
              if (Int32.TryParse(s, out num))
              {
                Console.WriteLine("Очередное число = {0}",
                                  num);
                // Добавляем полученное число в сумму
                sum += num;
              }
              // В случае ошибки переходим к следующему числу
            }
          }
        }

        // Вывод суммы
        Console.WriteLine("Сумма = {0}", sum);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
      }

      // Здесь находится тело метода IsAllDigits
    }
}
```

Программа `ParseSequenceWithSplit` начинает работу со считывания строки с клавиатуры. Затем методу `Split()` передается массив символов `dividers`, представляющих собой символы-разделители, использующиеся при отделении отдельных чисел в строке.

Далее программа циклически проходит по всем "подмассивам", созданным функцией `Split()`, применяя для этого цикл `foreach`. Программа пропускает все подстроки нулевой длины, а для непустых строк вызывает функцию `IsAllDigits()`, чтобы убедиться, что строка представляет собой число. Корректные строки преобразуются в целые числа и суммируются с аккумулятором `sum`. Некорректные числа игнорируются (я предпочел не генерировать сообщения об ошибках). Вот как выглядит типичный вывод данной программы:

```
Введите последовательность целых чисел, разделенных запятыми:
1, 2, a, 3, 4
Очередное число = 1
Очередное число = 2
Очередное число = 3
Очередное число = 4
Сумма = 10
Нажмите <Enter> для завершения программы...
```

Программа проходит по списку, рассматривая запятые, пробелы (или и то, и другое) как разделительные символы. Она пропускает "число" `a` и выводит общую сумму корректных чисел, равную 10. В реальных программах, однако, вряд ли можно просто так игнорировать некорректные числа, никак не сообщая об этом пользователю. При появлении во входном потоке любой программы "мусора" обычно требуется тем или иным способом обратить на это внимание пользователя.

Объединение массива строк в одну строку

Класс `String` имеет также метод `Join()`. Если у вас есть массив строк, можно использовать `Join()` для конкатенации всех строк. Можно также указать, что между строками в массиве следует вставить определенные символы:

```
string[] brothers = { "Chuck", "Bob", "Steve", "Mike" };
string theBrothers = string.Join(":", brothers);
```

В результате получится строка `theBrothers`, которая представляет собой `"Chuck:Bob:Steve:Mike"` (имена в ней разделены двоеточием). Можно было использовать в качестве разделителя любой другой символ — `", ", "\t", " "` (первый элемент представляет собой запятую с последующим пробелом, второй — символ табуляции, а третий — несколько пробелов подряд).

Управление выводом программы

Управление выводом программы — важный аспект работы со строками. Подумайте сами: вывод программы — это именно то, что видит пользователь. Не имеет значения, насколько элегантны внутренняя логика и реализация программы — это вряд ли впечатлит пользователя; куда важнее для него корректность и внешнее представление выводимых программой данных.

Класс `String` предоставляет программисту ряд методов для форматирования выводимой строки. В следующих разделах будут рассмотрены такие методы, как `Trim()`, `Pad()`, `PadRight()`, `PadLeft()`, `Substring()` и `Concat()`.

Использование методов `Trim()` и `Pad()`

Из раздела "Удаление пробельных символов" вы узнали, как пользоваться методом `Trim()` и его более специализированными вариантами `TrimFront()` и `TrimEnd()`. В данном разделе обсуждается другой распространенный метод форматирования выходных данных. Можно использовать методы `Pad`, которые добавляют символы к любому из концов строки, чтобы расширить строку до некоторой заранее заданной длины. Например, можно добавить пробелы слева

или справа от строки, чтобы соответствующим образом выровнять ее содержимое; можно добавить символы "*" и сделать еще множество подобных вещей. Приведенная далее небольшая программа AlignOutput использует Trim() и Pad() для обрезки и выравнивания ряда имен (код, связанный с Trim() и Pad(), отображен полужирным шрифтом):

```
using System;
using System.Collections.Generic;
// AlignOutput - выравнивание множества строк
// для улучшения внешнего вида вывода программы
namespace AlignOutput
{
  class Program
  {
    public static void Main(string[] args)
    {
      List<string> names = new List<string> {"Christa ",
                                             " Sarah",
                                             "Jonathan",
                                             "Sam",
                                             " Schmekowitz "
                                            };
      // Выводим имена.
      Console.WriteLine("Следующие имена имеют разные длины:");

      foreach (string s in names)
      {
        Console.WriteLine("Имя '" + s + "' до обработки");
      }

      Console.WriteLine();
      // Делаем строки выровненными влево, одинаковой длины
      // Сначала копируем исходный массив
      List<string> stringsToAlign = new List<string>();

      // Удаляем лишние пробелы с обоих концов
      for (int i = 0; i < names.Count; i++)
      {
        string trimmedName = names[i].Trim();
        stringsToAlign.Add(trimmedName);
      }

      // Находим длину самой длинной строки
      int maxLength = 0;

      foreach (string s in stringsToAlign)
      {
        if (s.Length > maxLength)
        {
          maxLength = s.Length;
        }
      }
```

```
      // Выравниваем все строки, приводя их
      // к максимальной длине
      for (int i = 0; i < stringsToAlign.Count; i++)
      {
        stringsToAlign[i] =
            stringsToAlign[i].PadRight(maxLength+1);
      }

      // Выводим получившиеся строки
      Console.WriteLine("Те же имена выровнены и " +
                        "имеют одинаковую длину:");

      foreach (string s in stringsToAlign)
      {
        Console.WriteLine("Имя '" + s + "' после обработки");
      }

      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
    }
  }
}
```

`AlignOutput` определяет список имен `List<string>` с разными выравниванием и длиной (вы можете переписать программу так, чтобы эти имена считывались с клавиатуры или из файла). Функция `Main()` сначала выводит эти имена на экран в том виде, в котором они получены программой. Затем вызываются методы `Trim()` и `PadRight()`, существенно улучшающие внешний вид выводимых программой строк:

```
Следующие имена имеют разные длины:
Имя 'Christa ' до обработки
Имя ' Sarah' до обработки
Имя 'Jonathan' до обработки
Имя 'Sam' до обработки
Имя ' Schmekowitz ' до обработки

Те же имена выровнены и имеют одинаковую длину:
Имя 'Christa     ' после обработки
Имя 'Sarah       ' после обработки
Имя 'Jonathan    ' после обработки
Имя 'Sam         ' после обработки
Имя 'Schmekowitz ' после обработки
```

Процесс выравнивания начинается с создания копии переданного ему массива `names`. Код начинается с цикла, вызывающего `Trim()` для каждого элемента массива, чтобы удалить лишние пробельные символы с обоих концов строки. Затем выполняется второй цикл, в котором происходит поиск самого

длинного элемента массива. И наконец, в последнем цикле для элементов массива вызывается метод `PadRight()`, удлиняющий строки и делающий их равными по длине.

Метод `PadRight(10)` увеличивает строку так, чтобы ее длина составляла как минимум 10 символов. Например, если длина исходной строки — 6 символов, то метод `PadRight(10)` добавит к ней справа 4 пробела.

Наконец, код проходит по полученному списку строк, выводя их на экран. Вот и все.

Использование метода `Concat()`

Зачастую программисты сталкиваются с задачей разбиения строки или вставки некоторой подстроки в середину другой строки. Заменить один символ другим проще всего с помощью метода `Replace()`:

```
string s = "Danger NoSmoking";
s = s.Replace(' ', '!')
```

Этот фрагмент исходного текста преобразует начальную строку в `"Danger!NoSmoking"`. Замена всех вхождений одного символа (в данном случае — пробела) другим (восклицательным знаком) особенно полезна при генерации списка элементов, разделенных запятыми для упрощения анализа. Однако более распространенный и сложный случай предусматривает разбиение единой строки на подстроки, отдельную работу с каждой подстрокой с последующим их объединением в единую модифицированную строку.

Приведенная далее демонстрационная программа `RemoveWithSpace` использует метод `Concat()` для удаления из строки пробельных символов (пробелов, символов табуляции и новой строки).

```
using System;
// RemoveWhiteSpace - удаление символов из предопределенного
// множества из заданной строки.
namespace RemoveWhiteSpace
{
  public class Program
  {
    public static void Main(string[] args)
    {
      // Определение множества пробельных символов
      char[] whiteSpace = {' ', '\n', '\t'};
      // Начинаем работу со строкой, в которой имеются
      // пробельные символы
      string s = " this is a\nstring";
      Console.WriteLine("До:" + s);
      // Выводим строку с удаленными пробельными символами
      Console.Write("После:");
```

```
      // Поиск пробельных символов
      for (;;)
      {
        // Ищем позиции искомых символов; если таковых в
        // строке больше нет, выходим из цикла
        int offset = s.IndexOfAny(whiteSpace);

        if (offset == -1)
        {
          break;
        }

        // Разбиваем строку на две части — до найденного
        // символа и после него.
        string before = s.Substring(0, offset);
        string after = s.Substring(offset + 1);

        // Объединяем эти части, но уже без найденного
        // символа.
        s = String.Concat(before, after);

        // Циклически ищем следующий пробельный символ
        // в модифицированной строке s
      }

      Console.WriteLine(s);

      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
    }
  }
}
```

Ключевым в этой демонстрационной программе является код, выделенный полужирным шрифтом. Он циклически удаляет из строки вхождения всех символов, содержащихся в массиве `whiteSpace`.

Цикл использует `IndexOfAny()` для поиска первого вхождения какого-либо из символов из массива `whiteSpace`. Выход из цикла не осуществляется до тех пор, пока из исходной строки не будут удалены все эти символы. Метод `IndexOfAny()` возвращает индекс первого найденного символа из исходного массива в строке. Возвращаемое значение `-1` указывает, что в строке нет элементов из переданного массива.

Первая итерация цикла находит ведущий пробел в целевой строке, индекс которого, равный `0`, возвращается методом `IndexOfAny()`. Первый вызов `Substring()` возвращает пустую строку, а второй — всю строку после пробела. Затем функция `Concat()` объединяет эти строки, создавая строку без начального пробела.

Вторая итерация находит и удаляет пробел после слова "this", давая в результате строку "thisis a\nstring". На третьей и четвертой итерациях находятся и удаляются пробел и символ \n. После этого очередной вызов IndexOfAny() не находит ни одного искомого символа и возвращает -1, на чем работа цикла и завершается.

Демонстрационная программа сначала выводит строку, содержащую пробельные символы, затем удаляет их и выводит получившуюся в результате строку:

```
До: this is a
string
После:thisisastring
Нажмите <Enter> для завершения программы...
```

Использование метода Split()

В программе RemoveWhiteSpace было продемонстрировано применение методов Concat() и IndexOf(); однако использованный способ решения поставленной задачи не самый эффективный. Стоит немного подумать, и можно получить существенно более эффективную функцию с использованием уже знакомой функции Split(). Вот код метода, выполняющего необходимые действия:

```
// RemoveSpecialChars удаляет из строки все указанные
// символы
public static string RemoveSpecialChars(string input,
                                         char[] targets)
{
    // Разбиение входной строки с использованием символов
    //  targets в качестве разделителей
    string[] subStrings = input.Split(targets);

    // Строка output будет содержать выходную информацию
    string output = "";

    // Цикл по всем подстрокам, полученным при вызове Split()
    foreach(string subString in subStrings)
    {
        output = String.Concat(output, subString);
    }
    return output;
}
```

В этой версии для разбиения входной строки на множество подстрок используется функция Split() с удаляемыми символами в качестве символов-разделителей. Поскольку разделители не включаются в подстроки, создается эффект их удаления. Такая логика проста и менее подвержена ошибкам при реализации.

Цикл `foreach` в этой версии функции собирает части строки в единое целое с использованием метода `Concat()`. Вывод программы остается неизменным, но вынос кода в отдельный метод делает программу проще и понятнее.

Форматирование строк

Класс `String` предоставляет в распоряжение программиста метод `Format()` для форматирования вывода, в основном — чисел. В своей простейшей форме `Format()` позволяет вставлять строки, числа, логические значения в середину форматируемой строки. Рассмотрим, например, следующий вызов:

```
string myString = String.Format("{0} * {1} = {2}", 2, 5, 2*5);
```

Первый аргумент метода `Format()` — *форматная строка* (строка формата). Элементы `{n}` в ней указывают, что *n*-й аргумент, следующий за форматной строкой, должен быть вставлен в этой точке. `{0}` означает первый аргумент (в данном случае — 2), `{1}` — второй (3) и т.д. В приведенном фрагменте получившаяся строка присваивается переменной `myString` и имеет следующий вид:

```
"2 * 5 = 10"
```

Если не указано иное, метод `Format()` для каждого типа аргумента использует формат по умолчанию. Для указания формата вывода в фигурных скобках, кроме номера аргумента, можно размещать дополнительные модификаторы, которые показаны в табл. 3.1. Например, `{0:E6}` гласит "Вывод числа в экспоненциальном виде с использованием шести знакомест в дробной части".

Таблица 3.1. Модификаторы, используемые функцией `String.Format()`

Модификатор	Пример	Результат	Примечание
`C` — денежные единицы	`{0:C}` для `123.456`	`$123.45`	Символ валюты зависит от настройки региональных стандартов
	`{0:C}` для `-123.456`	`($123.45)`	
`D` — десятеричное число	`{0:D5}` для `123`	`00123`	Только для целых чисел
`E` — число в экспоненциальной форме	`{0:E}` для `123.45`	`1.2345E+002`	Известна также как научная запись

Окончание табл. 3.1

Модификатор	Пример	Результат	Примечание
F — число с плавающей точкой	{0:F2} для 123.4567	123.45	Число после F указывает количество цифр после десятичной точки
N — число	{0:N} для 123456.789	123,456.79	Добавляет запятые и округляет число до ближайших сотых
	{0:N1} для 123456.789	123,456.8	Указывает количество цифр после десятичной точки
	{0:N0} для 123456.789	123,457	Указывает количество цифр после десятичной точки
X — шестнадцатеричное число	{0:X} для 123	0x7B	Шестнадцатеричное число 7B равно десятичному числу 123. Применяется только для целых чисел
{0:0...}	{0:000.00} для 12.3	012.30	Вносит 0 там, где нет реальных цифр
{0:#...}	{0:###.##} для 12.3	12.3	Вносит пробелы; полезен при выравнивании по десятичной точке
	{0:##0.0#} для 0	0.0	Комбинация 0 и # заставляет вносить пробелы на местах # и обеспечивает наличие как минимум одной цифры, даже если число равно 0
{0:# или 0%}	{0:#00.#%} для .1234	12.3%	% заставляет выводить число как проценты (умножая на 100 и добавляя символ %)
	{0:#00.#%} для .0234	02.3%	

СОВЕТ

Метод `Console.WriteLine()` использует такую же систему замещения. Первый элемент, `{0}`, означает первую переменную или значение после форматной строки, и т.д. Получая те же аргументы, что и метод `Format()`, `Console.WriteLine()` выводит получившуюся строку на консоль. В этом методе можно использовать те же спецификаторы формата, которые применяются в функции `Format()`. Далее при необходимости вывести что-то на экран будет использоваться именно этот метод.

Все эти модификаторы могут показаться слишком запутанными, но вы всегда можете получить информацию о них в справочной системе C#. Чтобы увидеть модификаторы в действии, взгляните на приведенную далее демонстрационную программу `OutputFormatControls`, позволяющую ввести не только число с плавающей точкой, но и модификатор формата, который будет использован при выводе введенного числа обратно на экран:

```
// OutputFormatControls позволяет пользователю посмотреть,
// как влияют модификаторы форматирования на вывод чисел.
// Модификаторы вводятся в программу так же, как и числа,
// - в процессе работы программы
namespace OutputFormatControls
{
  using System;

  public class Program
  {
    public static void Main(string[] args)
    {
      // Бесконечный цикл для ввода чисел, пока пользователь
      // не введет вместо числа пустую строку, что является
      // сигналом к окончанию работы программы
      for(;;)
      {
        // Ввод числа и выход из цикла, если введена пустая
        // строка
        Console.WriteLine("Введите число с плавающей точкой");
        string numberInput = Console.ReadLine();
        if (numberInput.Length == 0)
        {
          break;
        }
        double number = Double.Parse(numberInput);

        // Ввод модификаторов форматирования, разделенных
        // пробелами
        Console.WriteLine("Введите модификаторы " +
                          "форматирования, разделенные " +
                          "пробелами");
        Console.WriteLine("(Например: C E F1 N0 0000000.00000)");
```

```
        char[] separator = {' '};
        string formatString = Console.ReadLine();
        string[] formats = formatString.Split(separator);

        // Цикл по введенным модификаторам
        foreach(string s in formats)
        {
          if (s.Length != 0)
          {
            // Создание управляющего элемента форматирования
            // из введенного модификатора
            string formatCommand = "{0:" + s + "}";

            // Вывод числа с применением созданного
            // управляющего элемента форматирования
            Console.Write(
                  "Модификатор {0} дает ", formatCommand);
            try
            {
              Console.WriteLine(formatCommand, number);
            }
            catch(Exception)
            {
              Console.WriteLine("<Неверный модификатор>");
            }
            Console.WriteLine();
          }
        }

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
      }
    }
}
```

Программа `OutputFormatControls` считывает вводимые пользователем числа с плавающей точкой в переменную `numberInput` до тех пор, пока не будет введена пустая строка, что является признаком окончания ввода. Обратите внимание на то, что для простоты программа не выполняет никаких тестов для проверки корректности введенного числа с плавающей точкой.

Затем программа считывает ряд модификаторов форматирования, разделенных пробелами. Каждый из них далее комбинируется со строкой `{0}` в переменной `formatCommand`. Например, если вы ввели **N4**, программа создаст управляющий элемент `{0:N4}`, после чего введенное пользователем число будет выведено на экран с применением этого элемента:

```
Console.WriteLine(formatCommand, number);
```

В рассмотренном только что случае модификатора `N4` команда, по сути, превращается в

```
Console.WriteLine("{0:N4}", number);
```

Вот как выглядит типичный вывод программы на экран (полужирным шрифтом выделен ввод пользователя):

```
Введите число с плавающей точкой
12345.6789
Введите модификаторы форматирования, разделенные пробелами
(Например: C E F1 N0 0000000.00000)
C E F1 N0 0000000.00000
Модификатор {0:C} дает $12,345.68
Модификатор {0:E} дает 1.234568E+004
Модификатор {0:F1} дает 12345.7
Модификатор {0:N0} дает 12,346
Модификатор {0:0000000.00000} дает 0012345.67890
Введите число с плавающей точкой
.12345
Введите модификаторы форматирования, разделенные пробелами
(Например: C E F1 N0 0000000.00000)
00.0%
Модификатор {0:00.0%} дает 12.3%
Введите число с плавающей точкой

Нажмите <Enter> для завершения программы...
```

Будучи примененным к числу `12345.6789`, модификатор `N0` добавляет в нужное место запятую (часть `N`) и убирает все цифры после десятичной точки (часть `0`), что дает строку `12,346` (последняя цифра — результат округления, а не отбрасывания).

Аналогично, будучи примененным к числу `0.12345`, модификатор `00.0%` дает `12.3%`. Знак `%` приводит к умножению числа на 100 и добавлению символа `%` к выводимому числу. `00.0` указывает, что в выводимой строке должно быть по меньшей мере две цифры слева от десятичной точки и только одна — справа. Если тот же модификатор применить к числу `0.01`, будет выведена строка `01.0%`.

Непонятная (пока что) конструкция `try...catch` предназначена для перехвата всех потенциальных ошибок при вводе некорректных чисел. Однако об этом рассказывается совсем в другой главе, а именно — в главе 9, "Эти исключительные исключения".

StringBuilder: эффективная работа со строками

Построение длинных строк из набора коротких может стоить вам ваших рук, локтей и глаз... Как я уже говорил в начале этой главы, строка, будучи созданной, изменяться не может, так что код

```
string s1 = "rapid";
string s2 = s1 + "ly"; // s2 = rapidly.
```

на самом деле не добавляет "ly" к s1. Он создает новую строку, составленную из этих частей (s1 при этом остается неизменной). Другие операции, которые, как кажется, изменяют строки (например, Substring() или Replace()), на самом деле поступают так же.

Результат каждой операции над строкой представляет собой другую строку. Предположим, что надо соединить 1000 строк в одну большую строку. Вы создаете новую строку при помощи множества конкатенаций:

```
string[] listOfNames = ... // 1000 имен
string s = string.Empty;
for(int i = 0; i < 1000; i++)
{
    s += listOfNames[i];
}
```

Чтобы избежать такого большого количества модификаций строк, воспользуйтесь классом StringBuilder. Не забудьте только добавить в начале вашего исходного текста строку

```
using System.Text; // Где искать StringBuilder
```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В отличие от работы со String, действия с StringBuilder изменяют строку класса непосредственно, например:

```
StringBuilder builder = new StringBuilder("012");
builder.Append("34");
builder.Append("56");
string result = builder.ToString(); // result = 0123456
```

При создании экземпляр StringBuilder инициализируется существующей строкой. Если не указать начальное значение StringBuilder, будет использована пустая строка:

```
StringBuilder builder =
    new StringBuilder(); // До 16 символов по умолчанию
```

Можно создать StringBuilder с требуемой вам емкостью, что снизит накладные расходы на частое увеличение емкости для строки:

```
StringBuilder builder = new StringBuilder(256); // 256 символов.
```

Для добавления текста к концу текущего содержимого используется метод Append(). По завершении работы со строкой метод ToString() предоставит окончательный результат работы. Вот как выглядит StringBuilder-версия только что приведенного кода с циклом:

```
StringBuilder sb =
    new StringBuilder(20000);  // Выделение памяти
for(int i = 0; i < 1000; i++)
{
  sb.Append(listOfNames[i]);   // Тот же список имен
}
string result = sb.ToString(); // Получение результата
```

Класс StringBuilder имеет ряд других полезных методов, включая Insert(), Remove() и Replace(). Но в нем не хватает многих методов string, например Substring(), CopyTo() и IndexOf().

Предположим, например, что вы хотите перевести в верхний регистр только первый символ строки. При помощи StringBuilder сделать это существенно проще, чем описано в разделе "Преобразование символов строки в символы верхнего или нижнего регистра".

```
StringBuilder sb = new StringBuilder("jones");
sb[0] = char.ToUpper(sb[0]);
string fixedString = sb.ToString();
```

Здесь строка "jones" помещается в объект StringBuilder, выполняется обращение к первому символу строки StringBuilder как к sb[0], для его перевода в верхний регистр используется вызов метода char.ToUpper(), после чего символ в верхнем регистре вновь присваивается sb[0]. И наконец строка "Jones" получается из StringBuilder при помощи вызова метода ToString().

Представленный ранее пример BuildASentence может быть улучшен посредством применения в нем StringBuilder.

В части 2, "Объектно-ориентированное программирование на C#", рассматривается функциональная возможность C# — *методы расширения*. В приведенном в ней примере к классу String добавляется несколько удобных методов, а также описывается, как преобразовывать между собой такие типы, как String, массивы char и массивы byte. Все эти операции могут регулярно требоваться при повседневной работе.

Глава 4
Операторы

В ЭТОЙ ГЛАВЕ...

- Выполнение арифметических действий
- Логические операции
- Составные логические операторы

Математики создают переменные и выполняют над ними различные действия, складывая их, умножая, а иногда, представьте себе, даже интегрируя. В главе 2, "Работа с переменными", описано, как объявлять и определять переменные, но в ней ничего не говорится о том, как их использовать после объявления, чтобы получить что-то полезное. В этой главе рассматриваются операции, которые могут быть произведены над переменными. Для выполнения операций требуются *операторы*, такие как +, −, =, < или &. Здесь речь пойдет об арифметических, логических и других операторах.

Арифметика

Все множество арифметических операторов можно разбить на несколько групп: простые арифметические операторы, операторы присваивания и специальные операторы, присущие только программированию. После такого обзора арифметических операторов нужен обзор логических операторов, но о них речь пойдет несколько позже.

Простейшие операторы

С большинством из этих операторов вы должны были познакомиться еще в школе. Они перечислены в табл. 4.1. Обратите внимание на то, что в программировании для обозначения умножения используется звездочка (*), а не крестик (×).

Таблица 4.1. Простые операторы

Оператор	Значение
- (унарный)	Отрицательное значение
*	Умножение
/	Деление
+	Сложение
- (бинарный)	Вычитание
%	Деление по модулю

Большинство этих операторов являются *бинарными*, поскольку они выполняются над двумя значениями: одно из них находится слева от оператора, а другое — справа. Единственным исключением является унарный минус, который так же прост, как и остальные рассматриваемые здесь операторы:

```
int n1 = 5;
int n2 = -n1;  // Теперь значение n2 равно -5
```

Значение `n2` представляет собой отрицательное значение `n1`.

Оператор деления по модулю может быть вам незнаком. Деление по модулю аналогично получению остатка после деления. Так, `5%3` равно `2` (`5/3=1`, остаток — `2`), и значение `25%3` равно `1` (`25/3=8`, остаток — `1`).

Арифметические операторы (кроме деления по модулю) определены для всех типов переменных. Оператор же деления по модулю не определен для чисел с плавающей точкой, поскольку при делении значений с плавающей точкой не существует остатка.

Порядок выполнения операторов

Значения некоторых выражений могут оказаться непонятными. Например, рассмотрим следующее выражение:

```
int n = 5 * 3 + 2;
```

Что имел в виду написавший такую строку программист? Что надо умножить 5 на 3, а затем прибавить 2? Или сначала сложить 3 и 2, а результат умножить на 5?

Язык C# обычно выполняет операторы слева направо, при этом умножение выполняется до сложения — так что результатом приведенного примера будет `17`.

В представленном далее выражении язык C# вычисляет значение `n`, сначала деля `24` на `6`, а затем деля получившееся значение на `2`:

```
int n = 24 / 6 / 2
```

Однако у операторов есть своя *иерархия*, приоритеты или, проще говоря, свой порядок выполнения. C# считывает все выражение и определяет, какие операторы имеют наивысший приоритет и должны быть выполнены до операторов с меньшим приоритетом. Например, приоритет умножения выше, чем сложения. Во многих книгах этому вопросу посвящены целые главы, но сейчас не стоит забивать этим ни главу, ни свою голову.

Никогда не полагайтесь на то, что вы (или кто-то иной) помните приоритеты операторов. Нет никакого позора в том, чтобы явно указать подразумеваемый порядок выполнения выражения посредством скобок.

Значение следующего выражения совершенно очевидно и не зависит от приоритета операторов:

```
int n = (7 % 3) * (4 + (6 / 3));
```

Скобки перекрывают приоритеты операторов, явно указывая, как именно компилятор должен интерпретировать выражение. C# ищет наиболее глубоко вложенную пару скобок и вычисляет выражение в ней; в данном случае это `6/3`, что дает значение `2`. В результате получается

```
int n = (7 % 3) * (4 + 2); // 6 / 3 = 2
```

Затем C# продолжает поиск скобок и вычисляет значения в них, что приводит к выражению

```
int n = 1 * 6; // (4 + 2) = 6
```

Так что в конечном счете получается, что `n` равно `6`.

Оператор присваивания

Язык C# унаследовал одну интересную концепцию от C и C++: присваивание является бинарным оператором, возвращающим значение аргумента справа от него. Присваивание имеет тот же тип, что и оба аргумента (типы которых должны быть одинаковы). Этот новый взгляд на присваивание никак не влияет на выражения, с которыми вы уже сталкивались:

```
n = 5 * 3;
```

В данном примере `5*3` равно `15` и имеет тип `int`. Оператор присваивания сохраняет это `int`-значение справа в `int`-переменной слева и возвращает значение `15`. То, что он возвращает значение, позволяет, например, сохранить это значение еще в одной переменной, т.е. написать

```
m = n = 5 * 3;
```

Если имеется несколько присваиваний, то они выполняются справа налево. В приведенном выше выражении правый оператор присваивания сохраняет значение `15` в переменной `n` и возвращает `15`, после чего левый оператор присваивания сохраняет значение `15` в переменной `m` и возвращает `15` (это возвращенное значение в данном примере больше никак не используется).

Такое странное определение присваивания делает корректным, например, следующий причудливый фрагмент:

```
int n;
int m;
n = m = 2;
```

СОВЕТ

Старайтесь избегать цепочек присваиваний, поскольку они менее понятны человеку, читающему исходный текст программы. Всего, что может запутать человека, читающего исходный текст вашей программы (включая и лично вас), следует избегать, ибо любые неточности ведут к ошибкам.

Оператор инкремента

Среди всех выполняемых в программах сложений добавление 1 к переменной — наиболее распространенная операция:

```
n = n + 1; // Увеличение n на 1
```

C# расширяет множество простых операторов набором присваивающих операторов, построенных из арифметического оператора и оператора присваивания. Например, `n+=1;` эквивалентно `n=n+1;`.

Присваивающие версии операторов имеются почти для каждого бинарного оператора: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`. Детальную информацию о

присваивающих операторах можно почерпнуть из соответствующего раздела справочной системы.

Но и это недостаточно кратко, и в C# имеется еще более краткое обозначение этого действия — оператор инкремента:

```
++n; // Увеличение n на 1
```

Все три приведенных выражения функционально эквивалентны, т.е. все они увеличивают значение `n` на `1`.

Оператор инкремента достаточно странен, но еще больше странности придает ему то, что на самом деле имеется два оператора инкремента: `++n` и `n++`. Первый, `++n`, называется *префиксным*, а второй, `n++`, — *постфиксным*. Разница между ними довольно тонкая, но очень важная.

Вспомните, что каждое выражение имеет тип и значение. В следующем фрагменте исходного текста и `++n`, и `n++` имеют тип `int`:

```
int n;
n = 1;
int p = ++n;
n = 1;
int m = n++;
```

Чему равны значения `p` и `m` после выполнения этого фрагмента? (*Подсказка*: можно выбрать 1 или 2.) Оказывается, значение `p` равно `2`, а значение `m` — `1`, т.е. значение выражения `++n` — это значение `n` *после* увеличения, а значение `n++` равно значению `n` *до* увеличения. Значение самой переменной `n` в обоих вариантах равно `2`.

Эквивалентные операторы декремента, `n--` и `--n`, используются для замены выражения `n=n-1`. Они работают точно так же, как и операторы инкремента.

Логично ли логическое сравнение

Язык C# также предоставляет к услугам программиста целый ряд логических операторов сравнения, показанных в табл. 4.2. Эти операторы называются *логическими сравнениями* (logical comparisons), поскольку они возвращают результат сравнения в виде значения `true` или `false`, имеющего тип `bool`.

Таблица 4.2. Логические операторы сравнения

Оператор...	...возвращает `true`, если...
`a == b`	`a` имеет то же значение, что и `b`
`a > b`	`a` больше `b`

Окончание табл. 4.2

Оператор...	...возвращает `true`, если...
`a >= b`	`a` больше или равно `b`
`a < b`	`a` меньше `b`
`a <= b`	`a` меньше или равно `b`
`a != b`	`a` не равно `b`

Вот примеры использования логических сравнений:

```
int m = 5;
int n = 6;
bool b = m > n;
```

В этом примере переменной `b` присваивается значение `false`, поскольку `5` *не* больше, чем `6`.

Логические сравнения определены для всех числовых типов, включая `float`, `double`, `decimal` и `char`. Все приведенные ниже выражения корректны:

```
bool b;
b = 3 > 2;      // true
b = 3.0 > 2.0;  // true
b = 'a' > 'b';  // false - позже в алфавитном порядке
                // означает "больше"
b = 'A' < 'a';  // true  - прописная 'A' больше
                // строчной 'a'
b = 'A' < 'b';  // true  - все прописные буквы меньше всех
                // строчных
b = 10M > 12M;  // false
```

Операторы сравнения всегда дают в качестве результата величину типа `bool`. Операторы сравнения, отличные от ==, неприменимы к переменным типа `string` (не волнуйтесь, C# предлагает другие способы сравнения строк — см. главу 3, "Работа со строками").

Сравнение чисел с плавающей точкой

Сравнение двух чисел с плавающей точкой может легко оказаться не вполне корректным, так что здесь нужна особая осторожность. Рассмотрим следующее сравнение:

```
float f1;
float f2;
f1 = 10;
f2 = f1 / 3;
bool b1 = (3 * f2) == f1; // b1 равно true, если (3*f2) равно f1
```

```
f1 = 9;
f2 = f1 / 3;
bool b2 = (3 * f2) == f1;
```

Обратите внимание на то, что в строках 5 и 8 примера сначала содержится оператор присваивания =, а затем — оператор сравнения ==. Это разные операторы. C# сначала выполняет логическое сравнение, а затем присваивает его результат переменной слева от оператора присваивания.

Единственное отличие между вычислениями `b1` и `b2` состоит в исходном значении `f1`. Так чему же равны значения `bl` и `b2`? Очевидно, что значение `bl` равно `true`: 9/3 равно 3, 3*3 равно 9, 9 равно 9. Никаких проблем!

ВНИМАНИЕ!

Значение `bl` не столь очевидно: 10/3 равно 3.3333...; 3.3333...*3 равно 9.9999... Но равны ли числа 9.9999... и 10? Способ округления значений математическими операциями может повлиять на результат сравнения, что означает, что вам нужно проявлять осторожность, делая предположения о результатах математических операций. Несмотря на то что два значения вам представляются потенциально равными, для компьютера это не так. Следовательно, использование оператора == с числами с плавающей точкой обычно не рекомендуется ввиду возможности внесения ошибки в код.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Чтобы обойти вопросы округления и сравнить значения `f1` и `f2`, можно воспользоваться функцией для вычисления абсолютного значения следующим образом:

```
Math.Abs(f1-f2*3.0)<.00001; // ...или другая степень точности
```

Такая функция вернет значение `true` в обоих случаях. Вместо значения `.00001` можно использовать константу `Double.Epsilon`, чтобы получить максимальную точность. Эта константа представляет собой наименьшую возможную разницу между двумя неравными значениями типа `double`.

Чтобы узнать, какие еще возможности скрывает в себе класс `System.Math`, воспользуйтесь поиском **`math`** в справочной системе.

Составные логические операторы

Для переменных типа `bool` имеются специфичные для них операторы, показанные в табл. 4.3.

ЗАПОМНИ!

Оператор `!` (НЕ) представляет собой логический эквивалент знака "минус". Например, `!a` истинно, если `a` ложно, и ложно, если `a` истинно.

Таблица 4.3. Составные логические операторы

Оператор...	...возвращает `true`, если...
`!a`	`a` равно `false`
`a & b`	`a` и `b` равны `true`
`a | b`	Либо `a`, либо `b`, либо они обе равны `true` (`a` и/или `b`)
`a ^ b`	Либо `a`, либо `b`, но не обе одновременно равны `true` (либо `a`, либо `b`)
`a && b`	`a` и `b` равны `true` (сокращенное вычисление)
`a || b`	Либо `a`, либо `b`, либо они обе равны `true` (сокращенное вычисление)

Следующие два оператора также вполне просты и понятны. `a&b` истинно тогда и только тогда, когда и `a`, и `b` одновременно равны `true`; `a|b` истинно тогда и только тогда, когда или `a`, или `b`, или оба они одновременно равны `true`. Оператор `^` (*исключающее или*) возвращает значение `true` тогда и только тогда, когда значения `a` и `b` различны, т.е. когда одно из значений — `true`, а другое — `false`. Все перечисленные операторы возвращают в качестве результата значение типа `bool`.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Операторы `&`, `|` и `^` имеют версии, называющиеся *побитовыми* (bitwise). При применении к переменным типа `int` эти операторы выполняют действия с каждым битом отдельно. Таким образом, `6&3` равно `2` ($0110_2 \& 0011_2$ равно 0010_2), `6|3` равно `7` ($0110_2 | 0011_2$ равно 0111_2), а `6^3` равно `5` ($0110_2 \wedge 0011_2$ равно 0101_2). Бинарная арифметика — очень интересная вещь, но она в этой книге не рассматривается. Вы можете поискать информацию о ней в Интернете самостоятельно.

Последние два оператора очень похожи на предыдущие, но имеют одно едва уловимое отличие. В чем оно заключается, вы сейчас поймете. Рассмотрим следующий пример:

```
bool b = (ЛогическоеВыражение1) & (ЛогическоеВыражение2);
```

В этом случае C# вычисляет `ЛогическоеВыражение1` и `ЛогическоеВыражение2`, а затем смотрит, равны они оба `true` или нет, чтобы найти, какое значение следует присвоить переменной `b`. Но может оказаться, что C# выполняет лишнюю работу — ведь если одно из выражений равно `false`, то каким бы ни

было второе, результат все равно не может быть равным `true`. Тем не менее оператор `&` вычислит оба выражения.

Оператор `&&` позволяет избежать вычисления второго выражения, если после вычисления первого конечный результат очевиден:

```
bool b = (ЛогическоеВыражение1) && (ЛогическоеВыражение2);
```

В этой ситуации C# вычисляет значение `ЛогическоеВыражение1`, и если оно равно `false`, то переменной `b` присваивается значение `false` и `ЛогическоеВыражение2` не вычисляется. Если же `ЛогическоеВыражение1` равно `true`, то C# вычисляет `ЛогическоеВыражение2` и после этого определяет, какое значение присвоить переменной `b`. Оператор `&&` использует *сокращенное вычисление*, так как второе выражение вычисляется только при необходимости.

СОВЕТ

Большинство программистов используют оператор с двумя `&&`, а не с одним.

Оператор `||` работает аналогично, как видно из следующего выражения:

```
bool b = (ЛогическоеВыражение1) || (ЛогическоеВыражение2);
```

В этой ситуации C# вычисляет значение `ЛогическоеВыражение1`, и если оно равно `true`, то переменной `b` присваивается значение `true` и `ЛогическоеВыражение2` не вычисляется. Если же `ЛогическоеВыражение1` равно `false`, то C# вычисляет `ЛогическоеВыражение2` и после этого определяет, какое значение присвоить переменной `b`.

Вы можете называть эти операторы "сокращенное и" и "сокращенное или".

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Некоторые программисты полагаются на стандартные операторы для выполнения конкретных задач. Например, если выражение должно выполнить некоторое действие, а не только предоставить значение, не следует использовать сокращенный оператор, так как при этом C# может не выполнить второе действие при определенном результате первого действия. Пока что вам вряд ли стоит беспокоиться об этом, но лучше запомнить на будущее эту информацию.

Тип выражения

В вычислениях тип результата важен не менее самого результата. Рассмотрим следующее выражение:

```
int n;
n = (5 * 5) + 7;
```

Обычный калькулятор утверждает, что результат вычислений равен 32. Однако это выражение имеет не только значение, но и тип.

Будучи записанным на "языке типов", оно принимает следующий вид:

```
int [=] (int * int) + int;
```

Чтобы выяснить тип выражения, нужно следовать тому же шаблону, что и при вычислении его значения. Умножение имеет более высокий приоритет, чем сложение. Умножение `int` на `int` дает `int`. Далее следует сложение `int` и `int`, что в результате также дает `int`. Итак, вычисление типа приведенного выражения происходит таким образом:

```
(int * int) + int
int + int
int
```

Вычисление типа операции

Большинство операторов могут иметь несколько вариантов. Например, оператор умножения может быть следующих видов (стрелка здесь означает "производит"):

```
int     * int     ➪ int
uint    * uint    ➪ uint
long    * long    ➪ long
float   * float   ➪ float
decimal * decimal ➪ decimal
double  * double  ➪ double
```

Таким образом, `2*3` использует версию `int*int` оператора `*` и дает в результате `int 6`.

Неявное преобразование типов

Все хорошо, просто и понятно, если умножать две переменные типа `int` или две переменные типа `float`. Но что если типы аргументов слева и справа различны? Что, например, произойдет в следующей ситуации:

```
int anInt = 10;
double aDouble = 5.0;
double result = anInt * aDouble;
```

В C# нет оператора умножения `int*double`. Язык C# мог бы просто сгенерировать сообщение об ошибке и предоставить программисту возможность решать проблему самостоятельно. Однако он пытается понять намерения программиста и помочь ему. В C# есть операторы умножения `int*int` и `double*double`. Язык C# мог бы преобразовать `aDouble` в значение `int`, но такое преобразование привело бы к потере дробной части числа (цифр после десятичной точки). Поэтому вместо этого он преобразует `anInt` в значение

типа double и использует оператор умножения double*double. Это действие известно как *неявное повышение* типа (implicit promotion).

Такое повышение называется *неявным*, поскольку C# выполняет его автоматически, и является *повышением*, так как включает естественную концепцию высоты типа. Список операторов умножения был приведен в порядке повышения — от int до double или от int до decimal — от типа меньшего размера к типу большего размера. Между типами с плавающей точкой и decimal неявное преобразование не выполняется. Преобразование из более емкого типа, такого как double, в менее емкий, такой как int, называется *понижением* (demotion).

ВНИМАНИЕ!

Неявные понижения запрещены. В таких случаях C# генерирует сообщение об ошибке.

Явное преобразование типов

Но что если C# ошибается? Если на самом деле программист *хотел* выполнить целочисленное умножение? Вы можете изменить тип любой переменной с типом-значением с помощью оператора *приведения* типа (cast), который представляет собой требуемый тип, заключенный в скобки и располагаемый непосредственно перед приводимой переменной или выражением. Таким образом, в следующем выражении используется оператор умножения int*int:

```
int anInt = 10;
double aDouble = 5.0;
int result = anInt * (int)aDouble;
```

Приведение aDouble к типу int известно как *явное понижение* (explicit demotion) или *понижающее приведение* (downcast). Понижение является явным, поскольку программист явно объявил о своих намерениях.

ЗАПОМНИ!

Вы можете осуществить приведение между двумя любыми типами-значениями, независимо от их взаимной высоты.

СОВЕТ

Избегайте неявного преобразования типов. Делайте все изменения типов-значений явными с помощью оператора приведения — это снижает вероятность непреднамеренной ошибки и повышает удобочитаемость кода.

Оставьте логику в покое

Язык C# не позволяет преобразовывать другие типы в тип bool или выполнять преобразование типа bool в другие типы.

Типы при присваивании

Все сказанное о типах выражений применимо и к оператору присваивания.

ВНИМАНИЕ!

Случайные несоответствия типов, приводящие к генерации сообщений об ошибках, обычно происходят в операторах присваивания, а не в точке действительного несоответствия. Рассмотрим следующий пример умножения:

```
int n1 = 10;
int n2 = 5.0 * n1;
```

Вторая строка этого примера приведет к генерации сообщения об ошибке, связанной с несоответствием типов, но ошибка возникла *при присваивании*, а не при умножении. Вот что произошло: для того чтобы выполнить умножение, C# сначала неявно преобразовал `n1` в тип `double`, а затем выполнил умножение двух значений типа `double`. В результате получилось значение того же типа `double`.

Типы левого и правого аргументов оператора присваивания должны совпадать, но тип левого аргумента не может быть изменен. Поскольку C# не может неявно понизить тип выражения, компилятор генерирует сообщение о том, что он не может неявно преобразовать тип `double` в `int`. При использовании явного приведения никаких проблем не возникнет:

```
int n1 = 10;
int n2 = (int)(5.0 * n1);
```

(Скобки необходимы, потому что оператор приведения имеет очень высокий приоритет.) Такой исходный текст вполне работоспособен, так как *явное* понижение разрешено. Здесь значение `n1` будет повышено до `double`, выполнено умножение, а результат типа `double` будет понижен до `int`. Однако в этой ситуации необходимо задуматься о душевном здоровье программиста, поскольку написать просто `5*n1` было бы проще как для программиста, так и для C#.

Перегрузка операторов

Чтобы жизнь не казалась медом, знайте: поведение любого оператора можно изменить с помощью функциональной возможности C#, которая называется *перегрузкой оператора* (operator overloading). Перегрузка оператора, по существу, определяет новую функцию, которая выполняется в любой момент, когда вы используете оператор в проекте, где определена перегрузка. Перегрузка оператора на самом деле проще, чем кажется. Если вы пишете код

```
var x = 2+2;
```

то вы ожидаете, что x будет равно 4? Именно так и работает оператор +. Но все же на дворе XXI-й век! Чтобы сделать жизнь интереснее, давайте предоставим пользователям больше, чем они хотят, и при каждой операции сложения будем добавлять еще 1.

Для этого добавления 1 при каждой операции сложения необходимо создать пользовательский класс, который может использовать перегруженный оператор. Этот класс будет иметь некоторые пользовательские типы и метод, который будет использован для перегрузки операции. Короче говоря, при суммировании обычных чисел вы получите правильный ответ; если же вы добавляете специальные числа AddOne, то будет прибавлена лишняя единица:

```
public class AddOne
{
    public int x;
    public static AddOne operator +(AddOne a, AddOne b)
    {
        AddOne addone = new AddOne();
        addone.x = a.x + b.x + 1;
        return addone;
    }
}
```

После перегрузки оператор можно использовать как обычно:

```
public class Program {
    static void Main(string[] args) {
        AddOne foo = new AddOne();
        foo.x = 2;
        AddOne bar = new AddOne();
        bar.x = 3;
        // Теперь 2 + 3 равно 6...
        Console.WriteLine((foo + bar).x.ToString());
        Console.Read();
    }
}
```

В результате мы получаем не 5, а 6. Перегрузка оператора не является чем-то полезным для целых чисел, если только вы не планируете переписать законы математики. Однако, если у вас действительно есть сущности, для которых вы хотите иметь возможность суммирования, этот метод может оказаться полезным. Например, если у вас есть класс Product, вы можете переопределить оператор + для этого класса для добавления цены.

Глава 5

Управление потоком выполнения

В ЭТОЙ ГЛАВЕ...

- Что делать, *если*...
- Что делать *иначе*...
- Циклы `while` и `do...while`
- Использование `for` и область видимости

Рассмотрим следующую простую программу:

```
using System;
namespace HelloWorld
{
    public class Program
    {
        // Стартовая точка программы
        static void Main(string[] args)
        {
            // Приглашение для ввода имени
            Console.WriteLine("Введите ваше имя:");
            // Считывание введенного имени
            string name = Console.ReadLine();
            // Приветствие с использованием введенного имени
            Console.WriteLine("Привет, " + name);
            // Ожидание подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для "
                              "завершения программы...");
```

```
            Console.Read();
        }
    }
}
```

Толку от этой программы, помимо иллюстрации некоторых фундаментальных моментов программирования C#, очень мало. Она просто возвращает вам то, что вы ввели. Можно представить себе и более сложный пример программы, в которой выполняются некоторые вычисления над введенными данными и генерируется какой-то более сложный вывод на экран (иначе для чего проводить вычисления?..), но и эта программа будет очень ограничена в своей функциональности.

Одним из ключевых элементов любого компьютерного процессора является его возможность принимать решения. Под выражением "принимать решения" подразумевается, что процессор может пустить поток выполнения команд по тому или иному пути в зависимости от того, истинно или ложно некоторое условие. Любой язык программирования должен обеспечивать такую возможность управления потоком выполнения.

Три фундаментальных вида *управления потоком* (flow control) представляют собой инструкцию `if`, цикл и безусловный переход (один из циклов, `foreach`, будет рассмотрен в главе 6, "Глава для коллекционеров").

Ветвление с использованием `if` и `switch`

Основой принятия решения в C# является инструкция `if`:

```
if (Условие)
{
    // Этот код выполняется, если Условие истинно
}
// Этот код выполняется независимо от
// истинности Условия
```

Непосредственно за оператором `if` в круглых скобках содержится некоторое *условное выражение* типа `bool` (см. главу 2, "Работа с переменными"), после чего следует блок кода, заключенный в фигурные скобки. Если условное выражение истинно (имеет значение `true`), программа выполняет код в фигурных скобках. Если нет — этот код программой опускается. (Если программа выполняет код в фигурных скобках, то его выполнение завершается после закрывающей фигурной скобки и продолжается выполнение кода после нее.)

Работу оператора `if` проще понять, рассмотрев конкретный пример:

```
// Гарантируем, что a - неотрицательно:
// Если a меньше 0...
```

```
if (a < 0)
{
    // ...присваиваем этой переменной значение 0
    a = 0;
}
```

В этом фрагменте исходного текста проверяется, содержит ли переменная `a` отрицательное значение, и, если это так, переменной `a` присваивается значение `0`. Инструкция `if` гласит: "если `a` меньше нуля, присвоить переменной `a` значение `0`".

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Если в фигурные скобки заключена только одна инструкция, то их можно не использовать. C# рассматривает код `if(`*`условное выражение`*`) инструкция;` как если бы оно было записано как `if(`*`условное выражение`*`){инструкция;}`, т.е. в приведенном выше фрагменте можно было бы написать `if (a<0) a=0;`. Но для большей удобочитаемости лучше всегда использовать фигурные скобки.

Инструкция `if`

Рассмотрим небольшую программу, вычисляющую проценты. Пользователь вводит вклад и проценты, и программа подсчитывает сумму, получаемую по итогам года (это не слишком сложная программа). Вот как подобные вычисления выглядят на C#:

```
// Вычисление суммы вклада и процентов
decimal interestPaid;
interestPaid = principal * (interest / 100);
// Вычисление общей суммы
decimal total = principal + interestPaid;
```

В первом уравнении величина вклада `principal` умножается на величину процентной ставки `interest` (деление на `100` связано с тем, что пользователь вводит величину ставки в процентах). Получившаяся величина увеличения вклада сохраняется в переменной `interestPaid`, а затем суммируется с основным вкладом и сохраняется в переменной `total`.

Программа должна учитывать, что данные вводит всего лишь человек, которому свойственно ошибаться. Например, ошибкой должны считаться отрицательные величины вклада или процентов (конечно, в банке хотели бы, чтобы это было не так...). В приведенной далее программе `CalculateInterest` выполняются соответствующие проверки:

```
// CalculateInterest
// Вычисление величины начисленных процентов для данного
// вклада. Если процентная ставка или вклад отрицателен,
// генерируется сообщение об ошибке.
using System;
```

```
namespace CalculateInterest
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Приглашение для ввода вклада
            Console.Write("Введите сумму вклада:");
            string principalInput = Console.ReadLine();
            decimal principal =
                Convert.ToDecimal(principalInput);
            // Убеждаемся, что вклад не отрицателен
            if (principal < 0)
            {
                Console.WriteLine("Вклад не может "
                                  "быть отрицательным");
                principal = 0;
            }
            // Приглашение для ввода процентной ставки
            Console.Write("Введите процентную ставку:");
            string interestInput = Console.ReadLine();
            decimal interest =
                Convert.ToDecimal(interestInput);
            // Убеждаемся, что процентная ставка не
            // отрицательна
            if (interest < 0)
            {
                Console.WriteLine("Процентная ставка не "
                                  "может быть отрицательной");
                interest = 0;
            }
            // Вычисляем сумму величины процентных
            // начислений и вклада
            decimal interestPaid;
            interestPaid = principal * (interest / 100);
            // Вычисление общей суммы
            decimal total = principal + interestPaid;
            // Вывод результатов
            Console.WriteLine(); // Пропуск строки
            Console.WriteLine("Вклад = " + principal);
            Console.WriteLine("Проценты = "+interest+"%");
            Console.WriteLine();
            Console.WriteLine("Начисленные проценты = "
                              + interestPaid);
            Console.WriteLine("Общая сумма = " + total);
            // Ожидание реакции пользователя
            Console.WriteLine("Нажмите <Enter> для "
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

СОВЕТ

Программа `CalculateInterest` начинает свою работу с предложения пользователю ввести величину вклада. Это предложение выводится с помощью функции `WriteLine()`, которая выводит значение типа `string` на консоль. Всегда точно объясняйте пользователю, чего вы от него хотите. Если возможно, укажите также требуемый формат вводимых данных. Обычно на неинформативные приглашения наподобие одного символа > пользователи отвечают совершенно некорректно.

В программе для считывания всего пользовательского ввода до нажатия клавиши <Enter> в переменную типа `string` используется функция `ReadLine()`. Поскольку программа работает с величиной вклада как имеющей тип `decimal`, введенную строку следует преобразовать в переменную типа `decimal`, что и делает функция `Convert.ToDecimal()`. Полученный результат сохраняется в переменной `principal`.

ЗАПОМНИ!

Команды `ReadLine()`, `WriteLine()` и `ToDecimal()` служат примерами *вызовов методов*. Вызов метода делегирует некоторую работу другой части программы, именуемой методом. Подробно вызов метода будет описан в части 2, "Объектно-ориентированное программирование на C#", но приведенные здесь примеры очень просты и понятны. Если же вам что-то не ясно в вызовах функций, потерпите немного: ниже все будет детально объяснено.

В следующей строке проверяется переменная `principal`. Если она отрицательна, программа выводит сообщение об ошибке. Те же действия выполняются и для величины процентной ставки. После этого программа вычисляет общую сумму так, как было описано в начале раздела, и выводит конечный результат посредством нескольких вызовов функции `WriteLine()`.

Вот пример вывода программы при корректном пользовательском вводе:

```
Введите сумму вклада: 1234
Введите процентную ставку: 21

Вклад      = 1234
Проценты   = 21%

Начисленные проценты = 259.14
Общая сумма          = 1493.14
Нажмите <Enter> для завершения программы...
```

А так выглядит вывод программы при ошибочном вводе отрицательной величины процентной ставки:

```
Введите сумму вклада: 1234
Введите процентную ставку: -12.5
Процентная ставка не может быть отрицательной
```

```
Вклад      = 1234
Проценты   = 0%

Начисленные проценты = 0
Общая сумма          = 1234
Нажмите <Enter> для завершения программы...
```


СОВЕТ

Отступ внутри блока `if` повышает удобочитаемость исходного текста. C# игнорирует все отступы, но для человека они весьма важны. Большинство редакторов для программистов автоматически добавляют отступ при вводе оператора `if`. Для включения автоматического отступа в Visual Studio сначала выберите команду меню Tools⇨Options (Средства⇨Параметры), затем раскройте узел Text Editor (Текстовый редактор), после — C#, а в конце щелкните на вкладке Tabs (Табуляция). На ней включите отступы структуры и установите то количество пробелов на один отступ, которое вам по душе. Установите то же самое значение и в поле размера табуляции.

Инструкция `else`

Некоторые функции должны проверять взаимоисключающие условия. Например, в приведенном далее фрагменте исходного текста в переменной `max` сохраняется наибольшее из двух значений, `a` и `b`:

```
// Сохраняем наибольшее из двух значений,
// a и b, в переменной max
int max;
// Если a больше b...
if (a > b)
{
    // ...сохраняем значение a в переменной max
    max = a;
}
// Если a меньше или равно b...
if (a <= b)
{
    // ...сохраняем значение b в переменной max
    max = b;
}
```

Вторая конструкция `if` лишняя, поскольку проверяемые условия взаимоисключающие. Если `a` больше `b`, то `a` никак не может быть меньше или равно `b`. Для таких случаев в C# предусмотрено ключевое слово `else`, позволяющее указать блок, который выполняется, если не выполняется блок `if`. Вот как выглядит приведенный выше фрагмент кода при использовании `else`:

```
// Сохраняем наибольшее из двух значений, a и b,
// в переменной max
```

```
int max;
// Если a больше b...
if (a > b)
{
     // ...сохраняем значение a в переменной max;
    max = a;
}
else // в противном случае
{
     // ...сохраняем в переменной max значение b
    max = b;
}
```

Если a больше b, то выполняется первый блок; в противном случае выполняется второй блок. В результате в переменной max содержится наибольшее из значений a и b.

Как избежать else

При наличии нескольких else в исходном тексте можно легко запутаться, поэтому некоторые программисты предпочитают по возможности избегать использования else, если оно приводит к ухудшению удобочитаемости исходного текста. Так, рассмотренное выше вычисление максимального значения можно переписать следующим образом:

```
// Сохраняем наибольшее из двух значений, a и b,
// в переменной max
int max;
// Начнем с предположения, что a больше b
max = a;
// Если же это не так...
if (b > a)
{
    // ...то сохраняем в переменной max значение b
    max = b;
}
```

СОВЕТ

Программисты, считающие себя "крутыми", часто используют *тернарный оператор* ?:, однострочный эквивалент if/else:

```
bool informal = true;
string name = informal : "Chuck" ? "Charles"; // Вернет "Chuck"
```

Сначала вычисляется выражение до вопросительного знака. Если оно истинно, возвращается выражение после вопросительного знака, но до двоеточия. Если же оно ложно, возвращается выражение после двоеточия. Так конструкция if/else превращается в простое выражение. Но я бы советовал использовать этот не самый удобочитаемый оператор как можно реже.

Вложенные инструкции `if`

Программа `CalculateInterest` предупреждает пользователя о неверном вводе, но при этом продолжает вычислять начисленные проценты, несмотря на некорректность введенных значений. Вряд ли это правильное решение. Оно, конечно, не вызывает особых потерь процессорного времени, но только в силу простоты выполняемых программой подсчетов, в более же сложном случае это может привести к бо́льшим затратам. Кроме того, какой смысл запрашивать величину процентной ставки, если величина вклада к этому моменту уже введена неверно? Все равно результат придется проигнорировать, какое бы значение процентной ставки ни было введено. Программа должна запрашивать у пользователя величину процентной ставки только тогда, когда величина вклада введена верно, и выполнять вычисления тогда и только тогда, когда оба введенных значения корректны. Для этого необходимы две конструкции `if` — одна внутри другой.


ЗАПОМНИ!

Инструкция `if`, находящаяся в теле другой инструкции `if`, называется *встроенной* (embedded) или *вложенной* (nested). Приведенная далее программа `CalculateInterestWithEmbeddedTest` использует вложенную инструкцию `if` для того, чтобы избежать лишних вопросов при обнаружении некорректного ввода пользователя:

```
// CalculateInterestWithEmbeddedTest
//    Вычисление величины начисленных процентов для данного
//    вклада. Если процентная ставка или вклад отрицательный,
//    генерируется сообщение об ошибке и вычисления не
//    выполняются.
using System;

namespace CalculateInterestWithEmbeddedTest
{
  public class Program
  {
    public static void Main(string[] args)
    {
      // Определяем максимально возможное значение
      // процентной ставки
      int maximumInterest = 50;

      // Приглашение пользователю ввести величину исходного
      // вклада
      Console.Write("Введите сумму вклада:");
      string principalInput = Console.ReadLine();
      decimal principal = Convert.ToDecimal(principalInput);

      // Если исходный вклад отрицателен...
      if (principal < 0)
      {
```

```
            //...генерируем сообщение об ошибке...
            Console.WriteLine("Вклад не может быть отрицательным");
        }
        else  // Сюда попадаем, только если principal >= 0
        {
            // ...в противном случае просим ввести процентную
            // ставку
            Console.Write("Введите процентную ставку:");
            string interestInput = Console.ReadLine();
            decimal interest = Convert.ToDecimal(interestInput);

            // Если процентная ставка отрицательна или слишком
            // велика...
            if (interest < 0 || interest > maximumInterest)
            {
                // ...генерируем сообщение об ошибке
                Console.WriteLine("Процентная ставка не может "
                                  "быть отрицательной " +
                                  "или превышать "
                                  + maximumInterest);
                interest = 0;
            }
            else // Сюда мы попадаем, только если все в порядке
            {
                // И величина вклада, и процентная ставка
                // корректны — можно приступить к вычислению
                // вклада с начисленными процентами
                decimal interestPaid;
                interestPaid = principal * (interest / 100);

                // Вычисляем общую сумму
                decimal total = principal + interestPaid;

                // Выводим результат
                Console.WriteLine();  // skip a line
                Console.WriteLine("Вклад = "
                                  + principal);
                Console.WriteLine("Проценты = "
                                  + interest + "%");
                Console.WriteLine();
                Console.WriteLine("Начисленные проценты = "
                                  + interestPaid);
                Console.WriteLine("Общая сумма         = "
                                  + total);
            }
        }
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для "
                          "завершения программы...");
        Console.Read();
    }
  }
}
```

Программа начинает со считывания введенной пользователем величины исходного вклада. Если это значение отрицательно, она выводит сообщение об ошибке и завершает работу. Если же величина вклада не отрицательна, управление переходит к блоку `else`.

Проверка величины процентной ставки в этой программе несколько усовершенствована. Программа требует не только, чтобы введенное значение было не отрицательным, но и чтобы оно было меньше некоторого максимального значения. Применяемая в программе инструкция `if` использует следующий составной тест:

```
if (interest < 0 || interest > maximumInterest)
```

Выражение истинно, если `interest` меньше 0 или больше значения `maximumInterest`. Обратите внимание на то, что значение `maximumInterest` объявлено в начале программы, а не *жестко закодировано* в виде константы в исходном тексте условия. Жестко закодированное (прошитое) значение — это значение, которое указывается непосредственно в коде вместо создания константы для его хранения.

СОВЕТ

Определяйте важные константы в начале программы с использованием описательных символьных имен. Это позволяет легко находить и изменять константы. Если константа встречается в вашем коде десять раз и если она именованная, для ее изменения достаточно внести только одно изменение, а не десять.

Ввод корректной величины вклада и некорректной величины процентной ставки приводит к следующему выводу программы:

```
Введите сумму вклада: 1234
Введите процентную ставку: -12.5
Процентная ставка не может быть отрицательной или превышать 50.
Нажмите <Enter> для завершения программы...
```

Только при вводе корректных значений и вклада, и процентной ставки программа приступит к вычислениям и выведет интересующий вас результат:

```
Введите сумму вклада: 1234
Введите процентную ставку: 12.5

Вклад     = 1234
Проценты = 12.5%

Начисленные проценты = 154.250
Общая сумма          = 1388.250
Нажмите <Enter> для завершения программы...
```

Конструкция switch

Зачастую возникает необходимость сравнивать одну переменную с разными значениями. Пусть, например, переменная maritalStatus равна 0 для обозначения холостяков (незамужних), 1 — семейных, 2 — разведенных и 3 — вдов (вдовцов). Ну, и 4, если в анкете сказано, что это не наше дело... Для того чтобы по-разному отреагировать на различные значения этой переменной, можно воспользоваться серией инструкций if:

```
if (maritalStatus == 0)
{
    // Действия для холостяков
}
else
{
    if (maritalStatus == 1)
    {
        // Действия для семейных

        // ...и так далее...
```

Как видите, получается весьма неуклюжая конструкция. Поскольку такие проверки — не редкость в программистской практике, в C# имеется специальная конструкция для выбора из множества взаимоисключающих условий. Она называется switch и работает следующим образом:

```
switch(maritalStatus)
{
    case 0:
        // Действия для холостяков/незамужних
        break;
    case 1:
        // Действия для семейных
        break;
    case 2:
        // Действия для разведенных
        break;
    case 3:
        // Действия для вдов (вдовцов)
        break;
    case 4:
        // Действия для неизвестного семейного состояния
        break;
    default:
        // Действия, когда переменная принимает значение,
        // отличающееся от всех перечисленных выше (по всей
        // видимости, это означает, что произошла какая-то
        // ошибка)
        break;
}
```

Сначала вычисляется выражение в круглых скобках после ключевого слова switch. В данном случае это просто значение переменной maritalStatus. Затем вычисленное значение сравнивается со значениями каждого из операторов case. Если нужное значение не найдено, управление передается операторам, следующим за меткой default. Аргументом оператора switch может быть также строка string:

```
string s = "Davis";
switch(s)
{
    case "Mallory":
        // Некоторые действия
        break;
    case "Wells":
        // Некоторые действия
        break;
    case "Arturo":
        // Некоторые действия
        break;
    case "Brown":
        // Некоторые действия
        break;
    default:
        // Действия, если такой
        // фамилии нет в списке
}
```


ЗАПОМНИ!

При применении конструкции switch действует ряд ограничений.

- Аргумент инструкции switch() должен иметь перечислимый тип или тип string. Нельзя использовать числа с плавающей точкой.
- Значения case должны иметь тот же тип, что и аргумент инструкции switch.
- Значения case должны быть константами в том смысле, что их значения должны быть известны во время компиляции. (Инструкция наподобие case x некорректна, если x не является константой.)
- Каждая конструкция case должна завершаться оператором break (или какой-то иной командой выхода, например return). Оператор break передает управление за пределы конструкции switch.

  Допускается отсутствие break у case в том случае, когда несколько case приводят к одним и тем же действиям, т.е. одному блоку кода соответствует несколько case, как в следующем примере:

```
string s = "Davis";
switch(s)
{
    case "Davis":
```

```
        case "Hvidsten":
            // Действия для s, равного "Davis", те же,
            // что и для равного "Hvidsten"
            break;
        case "Smith":
            // ... действия для "Smith" ...
            break;
        default:
            // Действия, если такой фамилии нет в списке
            break;
}
```

Этот подход позволяет программе выполнять одни и те же действия как для строки `Davis`, так и для строки `Hvidsten`.

Циклы

Конструкция `if` позволяет программе выполняться различными путями в зависимости от результата вычисления значения типа `bool`. Она обеспечивает возможность создавать программы, существенно более интересные, чем те, которые могут быть написаны без ее использования. Еще одним применением машинной команды условного перехода является возможность *повторяющегося*, итеративного выполнения блока кода.

Рассмотрим еще раз программу `CalculateInterest` из раздела "Инструкция if" данной главы. Такие простые вычисления удобнее выполнять с помощью карманного калькулятора, чем писать для этого специальную программу.

Но что если необходимо вычислить проценты по вкладу для нескольких лет? Такая программа будет намного полезнее (конечно, простой макрос в Microsoft Excel все равно гораздо проще, чем требующаяся вам программа, но не стоит мелочиться). Итак, нужно выполнить некоторую последовательность инструкций несколько раз подряд. Это и есть *цикл* (loop).

Цикл `while`

Наиболее фундаментальный вид цикла создается с помощью ключевого слова `while` следующим образом:

```
while(Условие)
{
    // Код, повторно выполняемый до тех пор,
    // пока Условие не станет ложным
}
```

При первом обращении к циклу вычисляется условие в круглых скобках после ключевого слова `while`. Если оно истинно, выполняется следующий за

ним блок кода — тело цикла. По окончании выполнения тела цикла программа вновь возвращается к началу цикла и вычисляет условие в круглых скобках, и все начинается сначала. Если же в какой-то момент условие становится ложным, тело цикла не выполняется и управление передается коду, следующему за ним.


ЗАПОМНИ!

Если при первом обращении к циклу условие ложно, тело цикла не выполняется ни одного раза.


ВНИМАНИЕ!

Программисты зачастую косноязычны и могут не совсем корректно выражаться. Например, говоря о цикле `while`, они могут сказать, что тело цикла выполняется до тех пор, пока условие не станет ложным. Но такое определение не совсем корректно, так как можно решить, что выполнение цикла прервется в тот же момент, как только условие станет ложным Это не так. Программа не проверяет постоянно справедливость условия; проверка производится только тогда, когда управление передается в начало цикла.

Цикл `while` можно использовать для создания программы `CalculateInterestTable`, являющейся версией программы `CalculateInterest` с применением цикла. Она вычисляет таблицу величин вкладов по годам.

```
// CalculateInterestTable
// Вычисление величины начисленных процентов для данного
// вклада за определенный период
using System;
namespace CalculateInterestTable
{
  using System;

  public class Program
    {
    public static void Main(string[] args)
    {
      // Определяем максимально возможное значение
      // процентной ставки
      int maximumInterest = 50;

      // Приглашение пользователю ввести величину исходного
      // вклада
      Console.Write("Введите сумму вклада:");
      string principalInput = Console.ReadLine();
      decimal principal = Convert.ToDecimal(principalInput);

      // Если исходный вклад отрицательный...
      if (principal < 0)
      {
        //...генерируем сообщение об ошибке...
        Console.WriteLine("Вклад не может быть отрицательным");
```

```
}
else
{
  // ...в противном случае просим ввести процентную
  // ставку
  Console.Write("Введите процентную ставку:");
  string interestInput = Console.ReadLine();
  decimal interest = Convert.ToDecimal(interestInput);

  // Если процентная ставка отрицательна или слишком
  // велика...
  if (interest < 0 || interest > maximumInterest)
  {
    // ...генерируем сообщение об ошибке
    Console.WriteLine("Процентная ставка не может " +
                      "быть отрицательной " +
                      "или превышать " +
                      maximumInterest);
    interest = 0;
  }
  else
  {
    // И величина вклада, и процентная ставка
    // корректны — запрашиваем у пользователя срок,
    // для которого следует вычислить величины вкладов
    // с начисленными процентами
    Console.Write("Введите количество лет:");
    string durationInput = Console.ReadLine();
    int duration = Convert.ToInt32(durationInput);

    // Выводим введенные величины
    Console.WriteLine();  // Пропуск строки
    Console.WriteLine("Вклад = "
                      + principal);
    Console.WriteLine("Проценты = "
                      + interest + "%");
    Console.WriteLine("Срок      = "
                      + duration + " лет");
    Console.WriteLine();


    // Цикл по указанному пользователем количеству лет
    int year = 1;
    while(year <= duration)
    {
      // Вычисление вклада с начисленными процентами
      decimal interestPaid;
      interestPaid = principal * (interest / 100);

      // Вычисляем новое значение вклада
      principal = principal + interestPaid;

      // Округляем величину до центов
      principal = decimal.Round(principal, 2);
```

```
            // Выводим результат
            Console.WriteLine(year + "-" + principal);

            // Переходим к следующему году
            year = year + 1;
          }
        }
      }
      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
    }
  }
}
```

Вот как выглядит вывод программы `CalculateInterestTable`:

```
Введите сумму вклада: 1234
Введите процентную ставку: 12.5
Введите количество лет: 10

Вклад    = 1234
Проценты = 12.5%
Срок     = 10 лет

1-1388.25
2-1561.78
3-1757.00
4-1976.62
5-2223.70
6-2501.66
7-2814.37
8-3166.17
9-3561.94
10-4007.18
Нажмите <Enter> для завершения программы...
```

Каждое значение представляет общую сумму вклада по истечении указанного срока в предположении, что начисленные проценты добавляются к основному вкладу. Так, сумма 1234 доллара при ставке 12,5% за 9 лет превращается в 3561,94 доллара.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В большинстве значений для количества копеек выделяется две цифры. Однако в некоторых версиях C# завершающие нули могут не выводиться, и, например, сумма `12.70` может оказаться выведенной как `12.7`. Это поведение C# можно исправить с помощью специальных форматирующих символов, описываемых в главе 3, "Работа со строками" (C# версии 2.0 и более поздних выводят завершающие нули по умолчанию).

Программа `CalculateInterestTable` начинает работу со считывания величины вклада и процентной ставки и проверки их корректности. Затем она считывает количество лет, для которых надо подсчитать величины вкладов, и сохраняет их в переменной `duration`.

Прежде чем войти в цикл `while`, программа объявляет переменную `year`, инициализированную значением `1`. Эта переменная будет "текущим годом", т.е. ее значение будет увеличиваться с каждым выполнением тела цикла. Если номер года, хранящийся в переменной `year`, меньше общего количества лет, хранящегося в переменной `duration`, величина вклада для "этого года" вычисляется исходя из процентной ставки и величины вклада в "предыдущем году". Вычисленное значение программа выводит вместе со значением "текущего года".

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Инструкция `decimal.Round()` округляет вычисленное значение до центов.

Ключевая часть программы находится в последней строке тела цикла. Выражение `year = year + 1;` увеличивает переменную `year` на `1`. После увеличения значения года управление передается в начало цикла, где величина, хранящаяся в `year`, сравнивается с запрошенным количеством лет. В данном примере этот срок — 10 лет, так что, когда значение переменной `year` станет равным 11, т.е. превысит 10, программа передаст управление первой строке после цикла `while`, и работа цикла прекратится.

Переменная-счетчик `year` в программе `CalculateInterestTable` должна быть объявлена и инициализирована до цикла `while`, в котором она используется. Кроме того, переменная `year` должна увеличиваться (обычно в последней инструкции тела цикла). Как показано в примере, вы должны заранее позаботиться о том, какие переменные вам понадобятся в цикле. После того как вы напишете пару тысяч циклов `while`, все это будет делаться автоматически.

ВНИМАНИЕ!

При написании цикла `while` не забывайте увеличивать значение счетчика. Взгляните на приведенный пример исходного текста:

```
int nYear = 1;
while (nYear < 10)
{
    // ... Какой-то код ...
}
```

В этом примере переменная `nYear` не увеличивается. Без увеличения переменная `nYear` всегда содержит значение `1`, так что цикл работает вечно. Такая ситуация называется *зацикливанием* (бесконечным циклом, infinite loop). Единственный способ прекратить зацикливание — аварийно завершить программу извне.

ЗАПОМНИ!

Убедитесь в том, что условие прекращения работы цикла может быть достигнуто. Обычно это означает корректное увеличение значения счетчика цикла. В противном случае вы получите зацикливание программы, недовольных пользователей, падение продаж и много прочих неприятностей...

Цикл do...while

Разновидностью цикла `while` можно считать цикл `do...while`. При его использовании условие не проверяется, пока не будет достигнут *конец* цикла:

```
int year = 1;
do
{
    // ... Некоторые вычисления ...
    year = year + 1;
} while (year < duration);
```

В противоположность циклу `while` тело цикла `do...while` всегда выполняется по крайней мере один раз, независимо от значения переменной `duration`.

Операторы break и continue

Для управления циклом имеются два специальных оператора — `break` и `continue`. Оператор `break` вызывает прекращение выполнения цикла и передачу управления первому выражению непосредственно за циклом. Команда `continue` передает управление в начало цикла, к проверке его условия.

Предположим, вы хотите прекратить выполнение рассматривавшейся ранее программы, как только сумма вклада превысит начальную в некоторое заранее заданное число раз, независимо от того, какой срок прошел до этого момента. Это можно легко сделать, добавив в тело цикла следующие строки:

```
// Цикл по определенному количеству лет
int year = 1;

while(year <= duration)
{
    // Вычисление вклада с начисленными процентами
    decimal interestPaid;
    interestPaid = principal * (interest / 100);

    // Вычисляем новое значение вклада
    principal = principal + interestPaid;

    // Округляем величину до центов
    principal = decimal.Round(principal, 2);

    // Выводим результат
    Console.WriteLine(year + "-" + principal);
```

```
    // Переходим к следующему году
    year = year + 1;

    // Выясняем, достигли ли мы поставленной цели
    if (principal > (maxPower * originalPrincipal))
    {
        break;
    }

}
```

Оператор `break` не будет выполняться до тех пор, пока условие оператора `if` не станет истинным, т.е. пока вычисленная величина вклада не превысит исходную в `maxPower` раз. Оператор `break` передаст управление за пределы цикла `while(year<=duration)`, и выполнение программы продолжится с выражения, следующего непосредственно за этим циклом.

Цикл без счетчика

Программа `CalculateInterestTable` достаточно интеллектуальна для того, чтобы завершить работу, если пользователь ввел неверное значение вклада или процентной ставки. Однако трудно назвать дружественной программу, сразу же прекращающую работу, не давая пользователю ни одного шанса на исправление ошибки.

Комбинация `while` и `break` позволяет сделать программу немного более гибкой, что можно увидеть на примере исходного текста программы `CalculateInterestTableMoreForgiving`:

```
// CalculateInterestTableMoreForgiving
// Вычисление величины начисленных процентов для данного
// вклада за определенный период. Программа позволяет
// пользователю исправить ошибку ввода величины вклада и
// процентной ставки
using System;
namespace CalculateInterestTableMoreForgiving
{
  using System;

  public class Program
  {
    public static void Main(string[] args)
    {
      // Определяем максимально возможное значение
      // процентной ставки
      int maximumInterest = 50;

      // Приглашение пользователю ввести величину исходного
      // вклада; повторяем это приглашение до тех пор, пока
      // не будет получено корректное значение
      decimal principal;
```

```
while(true)
{
  Console.Write("Введите сумму вклада:");
  string principalInput = Console.ReadLine();
  principal = Convert.ToDecimal(principalInput);

  // Выход из цикла, если введенное значение корректно
  if (principal >= 0)
  {
    break;
  }

  // Генерируем сообщение о неверном вводе
  Console.WriteLine("Вклад не может быть " +
                    "отрицательным");
  Console.WriteLine("Повторите ввод");
  Console.WriteLine();
}

// Теперь вводим величину процентной ставки
decimal interest;
while(true)
{
  Console.Write("Введите процентную ставку:");
  string interestInput = Console.ReadLine();
  interest = Convert.ToDecimal(interestInput);

  // Если процентная ставка отрицательна или слишком
  // велика...
  if (interest >= 0 && interest <= maximumInterest)
  {
    break;
  }

  // ...генерируем сообщение об ошибке
  Console.WriteLine("Процентная ставка не может " +
                    "быть отрицательной " +
                    "или превышать " +
                    maximumInterest);
  Console.WriteLine("Повторите ввод");
  Console.WriteLine();
}

// И величина вклада, и процентная ставка
// корректны — запрашиваем у пользователя срок,
// для которого следует вычислить величины вкладов
// с начисленными процентами
Console.Write("Введите количество лет:");
string durationInput = Console.ReadLine();
int duration = Convert.ToInt32(durationInput);

// Выводим введенные величины
Console.WriteLine();  // Пропуск строки
Console.WriteLine("Вклад = " + principal);
```

```
      Console.WriteLine("Проценты = " +
                        interest + "%");
      Console.WriteLine("Срок      = " + duration +
                        " years");
      Console.WriteLine();


      // Цикл по указанному пользователем количеству лет
      int year = 1;
      while(year <= duration)
      {
        // Вычисление вклада с начисленными процентами
        decimal interestPaid;
        interestPaid = principal * (interest / 100);

        // Вычисляем новое значение вклада
        principal = principal + interestPaid;

        // Округляем величину до копеек
        principal = decimal.Round(principal, 2);

        // Выводим результат
        Console.WriteLine(year + "-" + principal);

        // Переходим к следующему году
        year = year + 1;
      }

      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
    }
  }
}
```

Данная программа во многом похожа на предыдущие примеры, за исключением исходного текста пользовательского ввода. Здесь оператор `if`, выявлявший неверный ввод пользователя, заменен циклом `while`:

```
decimal principal;
while(true)
{
  Console.Write("Введите сумму вклада:");
  string principalInput = Console.ReadLine();
  principal = Convert.ToDecimal(principalInput);

  // Выход из цикла, если введенное значение корректно
  if (principal >= 0)
  {
    break;
  }
```

```
    // Генерируем сообщение о неверном вводе
    Console.WriteLine("Вклад не может быть отрицательным");
    Console.WriteLine("Повторите ввод");
    Console.WriteLine();
}
```

В представленном фрагменте кода пользовательский ввод выполняется в цикле. Если введенное значение корректно, программа выходит из цикла и продолжает выполнение. Однако если во введенном значении имеется ошибка, пользователь получает сообщение о ней, и управление передается в начало цикла.

ЗАПОМНИ!

Программа выполняет цикл, пока пользователь не введет корректные данные (так что в наихудшем случае пользователь может постоянно вводить неверные данные, пока не умрет от старости).

Обратите также внимание на обращение условия, поскольку теперь проблема не в том, чтобы вывести сообщение об ошибке при некорректном вводе, а в том, чтобы завершить цикл при корректном. Проверка условия

```
principal < 0 || principal > maximumInterest
```

превратилась в проверку

```
interest >= 0 && interest <= maximumInterest
```

Понятно, что условие `interest>=0` противоположно условию `interest<0`. Менее очевидна замена оператора ИЛИ (`||`) оператором И (`&&`). Теперь оператор `if` гласит: "Выйти из цикла, если процентная ставка не меньше нуля И не больше максимального значения (другими словами, имеет корректную величину)". Обратите также внимание, что переменная `principal` должна быть объявлена за пределами цикла в соответствии с правилами видимости, которые объясняются в следующем разделе.

ВНИМАНИЕ!

Это может звучать как тавтология, но вычисление выражения `true` дает значение `true`. Таким образом, `while(true)` представляет собой бесконечный цикл, и от зацикливания спасает только наличие оператора `break` в теле цикла. Используя цикл `while(true)`, никогда не забывайте об операторе `break`, который должен прервать работу цикла по достижении заданного условия.

Вот как выглядит образец вывода программы:

```
Введите сумму вклада: -1000
Вклад не может быть отрицательным
Повторите ввод
```

```
Введите сумму вклада: 1000
Введите процентную ставку: -10
Процентная ставка не может быть отрицательной или превышать 50
Повторите ввод

Введите процентную ставку: 10
Введите количество лет: 5

Вклад    = 1000
Проценты = 10%
Срок     = 5 лет

1-1100.0
2-1210.00
3-1331.00
4-1464.10
5-1610.51
Нажмите <Enter> для завершения программы...
```

Программа отказывается принимать отрицательные значения вклада и процентов, позволяя пользователю исправить ошибку ввода.

ВНИМАНИЕ!

Всегда поясняйте пользователю, в чем именно он не прав, прежде чем предоставить ему возможность исправить допущенную ошибку. При ошибках, связанных с форматированием, неплохой идеей будет демонстрация примера корректного ввода. И будьте очень вежливы в своих сообщениях!

Правила области видимости

Переменная, объявленная в теле цикла, *определена только внутри* этого цикла. Рассмотрим следующий фрагмент исходного текста:

```
int days = 1;
while(days < duration)
{
    int average = value / days;
    // ... Некоторая последовательность операторов ...
    days = days + 1;
}
```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Переменная `average` не определена вне цикла `while`. Тому есть целый ряд причин, но рассмотрим одну из них. При первом выполнении цикла программа встречает объявление `int average`. При втором выполнении цикла то же объявление встречается еще раз, так что, если бы не было правила области видимости переменной, это привело бы к ошибке, так как переменная была бы уже определена. Можно привести и другие, более убедительные причины существования правила области видимости, но пока должно хватить и приведенного

аргумента. Достаточно сказать, что переменная `average` прекращает свое существование при достижении программой закрывающей фигурной скобки и вновь создается при каждом выполнении тела цикла.


СОВЕТ

Опытные программисты говорят, что *область видимости* переменной `average` ограничена циклом `while`.

Цикл `for`

Несмотря на свою простоту цикл `while` все же является вторым по распространенности циклом в программах на C#. Пальму первенства прочно удерживает цикл `for`, имеющий следующую структуру:

```
for(Выражение1; Условие; Выражение2)
{
    // ... тело цикла ...
}
```

По достижении цикла `for` программа сначала выполняет *Выражение1*. Затем она вычисляет *Условие* и, если оно истинно, выполняет тело цикла, которое заключено в фигурные скобки и следует сразу за оператором `for`. По достижении закрывающей скобки управление переходит *Выражению2*, после чего вновь вычисляется *Условие*, и цикл повторяется. Фактически определение цикла `for` можно переписать как следующий цикл `while`:

```
Выражение1;
while(Условие)
{
    // ... тело цикла ...
    Выражение2;
}
```

Пример

Возможно, вы лучше разберетесь, как работает цикл `for`, взглянув на конкретный пример:

```
// Некоторое выражение на C#
a = 1;
// Цикл
for(int year = 1; year < duration; year = year + 1)
{
    // ... тело цикла ...
}
// Здесь программа продолжается
a = 2;
```

Предположим, программа выполнила присваивание `a=1;`. После этого она объявляет переменную `year` и инициализирует ее значением `1`. Далее программа сравнивает значение `year` со значением `duration`. Если `year` меньше `duration`, выполняется тело цикла в фигурных скобках. По достижении закрывающей скобки программа возвращается к началу цикла и выполняет инструкцию `year=year+1`, перед тем как вновь перейти к проверке условия `year<duration`.

ВНИМАНИЕ!

Переменная `year` не определена вне области видимости цикла `for`, которая включает как тело цикла, так и его заголовок.

Зачем нужны разные циклы

Зачем в C# нужен цикл `for`, если в нем уже есть такой цикл, как `while`? Наиболее простой и напрашивающийся ответ — он не нужен, так как цикл `for` не может сделать ничего такого, чего нельзя было бы повторить с помощью цикла `while`.

Однако разделы цикла `for` повышают удобочитаемость исходных текстов, четко указывая три части, имеющиеся в каждом цикле: настройку, условие выхода и увеличение значения счетчика. Такой цикл проще не только для чтения и понимания, но и для проверки его корректности (вспомните, что основная ошибка при работе с циклом `while` — забытое увеличение значения счетчика или некорректный критерий завершения цикла). Не зря цикл `for` (и его "родственник" — цикл `foreach`) встречается в программах на порядок чаще других разновидностей циклов.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Так уж сложилось, что в основном в первой части цикла `for` выполняется инициализация переменной-счетчика, а в последней — ее увеличение. Но это не более чем традиция — C# не требует этого от программиста. В данных двух частях цикла `for` можно выполнять какие угодно действия, хотя, конечно же, для отхода от традиционной схемы, нужны серьезные основания.

В цикле `for` особенно часто используется оператор инкремента (который вместе с другими операторами был описан в главе 4, "Операторы"). Обычно приведенный ранее цикл `for` записывается как

```
for(int year = 1; year < duration; year++)
{
    // ... тело цикла ...
}
```

СОВЕТ

Почти всегда в цикле `for` применяется постфиксная форма оператора инкремента, хотя в данном случае функционально она идентична префиксной.[1]

У цикла `for` имеется одно правило, которое я не в состоянии пояснить: если условие в цикле отсутствует, считается, что оно равно `true`.[2] Таким образом, `for(;;)` — такой же бесконечный цикл, как и `while(true)`.

Вложенные циклы

Внутренний цикл может находиться в теле другого, внешнего цикла:

```
for(... некоторое условие ...)
{
    for(... некоторое другое условие ...)
    {
        // ... некоторые действия ...
    }
}
```

Внутренний цикл выполняется полностью при каждом выполнении тела внешнего цикла. Переменная-счетчик цикла (такая, как `year`), используемая во внутреннем цикле, является неопределенной вне области видимости внутреннего цикла.

ЗАПОМНИ!

Цикл, содержащийся внутри другого цикла, называется *вложенным* (nested). Вложенные циклы не могут "пересекаться", т.е. приведенный далее исходный текст некорректен:

```
do                  // Начало цикла do..while
{
    for( ... )      // Начало цикла for
    {
    } while( ... ) // Конец цикла do..while
}                   // Конец цикла for
```

[1] К счастью, современные компиляторы достаточно интеллектуальны, чтобы понять, что в данном случае возвращаемое значение не играет никакой роли, и не выполнять дополнительной работы по сохранению начального значения переменной при использовании постфиксной формы оператора. — *Примеч. ред.*

[2] Это правило легко пояснить тем, что, если бы отсутствие условия воспринималось как `false`, в таком цикле выполнялась бы исключительно первая часть заголовка, но не последняя и не тело цикла. — *Примеч. ред.*

ЗАПОМНИ!

Оператор `break` внутри вложенного цикла прекращает выполнение только этого вложенного цикла. В приведенном далее фрагменте исходного текста оператор `break` завершает работу цикла Б и возвращает управление циклу А:

```
// Цикл for А
for(... некоторое условие ...)
{
    // Цикл for Б
    for(... некоторое другое условие ...)
    {
        // ... некоторые действия ...
        if (Истинное условие)
        {
            break; // Выход из цикла Б, но не из цикла А
        }
    }
}
```

В C# нет команды `break`, которая обеспечивала бы одновременный выход из обоих циклов. Вы должны использовать два оператора `break`, по одному для каждого цикла.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Это не настолько уж большое ограничение, как может показаться. На практике зачастую сложную логику, содержащуюся внутри таких вложенных циклов, лучше инкапсулировать в виде функций. Выполнение оператора `return` в любом месте обеспечивает выход из функции, т.е. из всех циклов, какой бы ни была глубина вложенности. Функции и оператор `return` будут рассматриваться в главе 7, “Работа с коллекциями”.

Оператор `goto`

Управление может быть передано в другую точку при помощи оператора безусловного перехода `goto`. За этим оператором может следовать:

- метка;
- `case` в конструкции `switch`;
- ключевое слово `default`, обозначающее блок `default` в конструкции `switch`.

Два последних случая предназначены для перехода от одного блока `case` к другому в конструкции `switch`. Вот пример применения оператора `goto`:

```
// Если условие истинно...
if (a > b)
{
    // Управление оператором goto передается коду,
    // расположенному за меткой exitLabel
    goto exitLabel;
}
// Некоторый программный код
exitLabel:
// Управление передается в эту точку
```

Оператор `goto` крайне непопулярен по той же причине, по которой он является очень мощным средством: в силу его полной неструктурированности. Отслеживание переходов в нетривиальных ситуациях, превышающих несколько строк кода, — крайне неблагодарная задача. Это именно тот случай, когда говорят о "соплях" в программе.

СОВЕТ

Вокруг применения `goto` ведутся почти "религиозные войны". Доходит до критики C# просто за то, что в нем есть этот оператор. Но на самом деле в нем нет ничего ужасного или демонического. Другое дело что его применения следует избегать, если в этом нет крайней необходимости. Я бы рекомендовал использовать `goto` только *изредка* для связи двух `case` в конструкции `switch`:

```
switch(n) // Весьма надуманный пример, что и говорить
{
    case 0:
                // Что-то делаем для случая 0, затем...
        goto 3; // ...переходим к другому варианту,
                // не используя оператор break
    case 1:
                // Что-то делаем для случая 1
        break;
    case 3:     // Сюда осуществляется переход от случая 0
                // Здесь выполняются некоторые действия не
                // только для 3, но и для 0
        break;
    default:
                // Случай по умолчанию
        break;
}
```

Пожалуйста, не привыкайте использовать `goto`!

Глава 6

Глава для коллекционеров

В ЭТОЙ ГЛАВЕ...

- Массивы — переменные, хранящие много объектов
- Массивы и коллекции
- Инициализация и инициализаторы

Простые переменные, с которыми мы встречались раньше, предназначены для хранения единственного значения, и их возможностей недостаточно для того, чтобы, например, хранить десять чисел вместо одного. Язык C# предоставляет два вида переменных, которые могут хранить несколько объектов одновременно, — такие переменные обобщенно называются *коллекциями*. Двумя видами коллекции являются *массивы* (array) и более обобщенный *класс коллекции* (collection class).


ЗАПОМНИ!

Обычно, говоря "массив", я имею в виду именно массив, и если я говорю о классе коллекции, то подразумеваю именно его. Но когда я говорю о *коллекции* или *списке*, это может быть как массив, так и класс коллекции.

Массив представляет собой тип данных, хранящий список объектов, причем каждый из объектов имеет один и тот же тип: `int`, `double` и т.д.

Язык C# предоставляет обширную коллекцию коллекций (неплохой каламбур, правда?), таких как списки, очереди, стеки и пр. Большинство классов коллекций подобно массивам в том смысле, что они могут хранить однотипные объекты — или только яблоки, или только апельсины. Имеется в C# и несколько классов для хранения разнотипных объектов, но они могут быть полезными лишь в редких случаях.

Чтобы разобраться во всем материале книги, достаточно уметь работать с массивами и коллекцией `List` (хотя в этой главе вы встретитесь еще с двумя видами коллекций).

Массивы C#

В вашем распоряжении есть переменные, хранящие отдельные единственные значения. Классы могут использоваться для описания составных объектов. Но вам нужна еще одна конструкция для хранения множества объектов, например коллекции старинных автомобилей Билла Гейтса. Встроенный класс `Array` представляет собой структуру, которая может содержать последовательности однотипных элементов (чисел типа `int`, `double`, объектов `Vehicle` или `Motor` и т.п.; с такими объектами вы встретитесь в главе 7, "Работа с коллекциями").

Зачем нужны массивы

Рассмотрим задачу определения среднего из шести чисел с плавающей точкой. Каждое из этих шести чисел требует собственную переменную для хранения значения типа `double`:

```
double d0 = 5;
double d1 = 2;
double d2 = 7;
double d3 = 3.5;
double d4 = 6.5;
double d5 = 8;
```

Вычисление среднего этих переменных может выглядеть так, как показано далее (помните, что усреднение переменных типа `int` может привести к ошибкам округления, как было описано в главе 2, "Работа с переменными"):

```
double sum = d0 + d1 + d2 + d3 + d4 + d5;
double average = sum / 6;
```

Перечислять все элементы — очень утомительно, даже если их всего шесть. А теперь представьте, что необходимо усреднить 600 чисел или даже шесть миллионов...

Массив фиксированного размера

К счастью, вам не нужно именовать каждый из элементов. Язык C# предоставляет в распоряжение программиста массивы, которые могут хранить последовательности значений. Используя массив, вы можете разместить все значения типа `double` в одной переменной следующим образом:

```
double[] doublesArray = {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
```

Можно также объявить массив без инициализации, например

```
double[] doublesArray = new double[6];
```

Это объявление просто выделяет память для шести чисел типа `double`, не инициализируя их.


ЗАПОМНИ!

Класс `Array`, на котором основаны все массивы C#, использует специальный синтаксис, который делает его более удобным в применении. Квадратные скобки `[]` предоставляют доступ к отдельным элементам массива:

```
doublesArray[0] // Соответствует d0 (т.е. 5)
doublesArray[1] // Соответствует d1 (т.е. 2)
...
```

Нулевой элемент массива соответствует `d0`, первый — `d1` и т.д.


ЗАПОМНИ!

Номера элементов массива — 0, 1, 2, ... — известны как их *индексы.*


ЗАПОМНИ!

В C# индексы массивов начинаются с 0, а не с 1. Таким образом, элемент с индексом 1 не является первым элементом массива. Не забывайте об этом! *Первым является нулевой элемент массива!*

Использование `doublesArray` не привело бы к значительному улучшению, если бы в качестве индекса массива нельзя было использовать переменную. Применять цикл `for` существенно проще, чем записывать каждый элемент вручную, что и демонстрирует следующая программа:

```
// FixedArrayAverage
// Усреднение массива чисел фиксированного размера с
// использованием цикла
namespace FixedArrayAverage
{
  using System;

  public class Program
  {
```

```
    public static void Main(string[] args)
    {
      double[] doublesArray =
              {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};

      // Накопление суммы элементов
      // массива в переменной sum
      double sum = 0;
      for (int i = 0; i < 10; i++)
      {
        sum = sum + doublesArray[i];
      }

      // Вычисление среднего значения
      double average = sum / 10;
      Console.WriteLine(average);

      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
    }
  }
}
```

Программа начинает работу с инициализации переменной `sum` значением `0`. Затем программа циклически проходит по всем элементам массива `doublesArray` и прибавляет их к `sum`. По окончании цикла сумма всех элементов массива хранится в `sum`. Разделив ее на количество элементов массива, получаем искомое среднее значение, равное в данном случае 4,6 (можете проверить это с помощью своего калькулятора).

ПРОВЕРКА ГРАНИЦ МАССИВА

Программа `FixedArrayAverage` должна циклически проходить по массиву из 10 элементов. К счастью, цикл разработан так, что проходит ровно по 10 элементам массива. Ну а если бы была допущена ошибка и проход был сделан не по 10 элементам, а по иному их количеству? Рассмотрим два основных случая.

Что произойдет при выполнении 9 итераций? C# не трактует такую ситуацию как ошибочную. Если вы хотите рассмотреть только 9 из 10 элементов, то как C# может указывать вам, что именно вам нужно делать? Конечно, среднее значение при этом будет неверным, но программе это неизвестно.

Что произойдет при выполнении 11 или более итераций? В этом случае C# примет свои меры и не позволит индексу выйти за дозволенные пределы, чтобы вы не смогли случайно переписать какие-нибудь важные данные в па-

мяти. Чтобы убедиться в этом, измените сравнение в цикле `for`, заменив 10 значением 11:

```
for (int i = 0; i < 11; i++)
```

При выполнении программы вы получите диалоговое окно со следующим сообщением об ошибке:

```
IndexOutOfRangeException was unhandled
Index was outside the bounds of the array.
```

Здесь C# сообщает о произошедшей неприятности — исключении `IndexOutOfRangeException`, из названия которого и из поясняющего текста становится понятна причина ошибки: выход индекса за пределы допустимого диапазона. (Кроме того, выводится детальная информация о том, где именно и что произошло, но пока что вы не настолько знаете C#, чтобы разобраться в этом.)

Массив переменного размера

Массив, используемый в программе `FixedArrayAverage`, сталкивается с двумя серьезными проблемами:

- его размер фиксирован и равен 10 элементам;
- что еще хуже, значения этих элементов указываются непосредственно в тексте программы.

Значительно более гибкой была бы программа, которая могла бы считывать переменное количество значений, вводимое пользователем, ведь она могла бы работать не только с определенными в программе `FixedArrayAverage` значениями, но и с другими множествами значений. Формат объявления массива переменного размера несколько отличается от формата объявления массива фиксированного размера:

```
double[] doublesArrayVariable = new double[N];  // Переменный
double[] doublesArrayFixed    = new double[10]; // Постоянный
```

Здесь `N` — количество элементов в выделяемом массиве. Модифицированная версия программы `VariableArrayAverage` позволяет пользователю указать количество вводимых значений. Поскольку программа сохраняет введенные значения, она может не только вычислить среднее значение, но и вывести результат в удобном виде:

```
using System;
// VariableArrayAverage
// Вычисление среднего значения массива, размер которого
// указывается пользователем во время работы программы.
```

```
// Накопление введенных данных в массиве позволяет
// обращаться к ним неоднократно, в частности для генерации
// привлекательно выглядящего вывода на экран.
namespace VariableArrayAverage
{
  public class Program
  {
    public static void Main(string[] args)
    {
      // Сначала считывается количество чисел типа double,
      // которое пользователь намерен ввести для усреднения
      Console.Write("Введите количество усредняемых чисел: ");
      string numElementsInput = Console.ReadLine();
      int numElements = Convert.ToInt32(numElementsInput);
      Console.WriteLine();

      // Объявляем массив необходимого размера
      double[] doublesArray = new double[numElements];

      // Накапливаем значения в массиве
      for (int i = 0; i < numElements; i++)
      {
        // Приглашение пользователю для ввода чисел
        Console.Write("Введите число типа double №" +
                      (i + 1) + ": ");
        string val = Console.ReadLine();
        double value = Convert.ToDouble(val);

        // Вносим число в массив
        doublesArray[i] = value;
      }


      // Суммируем 'numElements' значений из массива в
      // переменной sum
      double sum = 0;
      for (int i = 0; i < numElements; i++)
      {
        sum = sum + doublesArray[i];
      }

      // Вычисляем среднее
      double average = sum / numElements;

      // Выводим результат на экран
      Console.WriteLine();
      Console.Write(average
                    + " является средним из ("
                    + doublesArray[0]);
      for (int i = 1; i < numElements; i++)
      {
        Console.Write(" + " + doublesArray[i]);
      }
```

```
        Console.WriteLine(") / " + numElements);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
  }
}
```

Вот как выглядит вычисление среднего для пяти последовательных чисел от 1 до 5:

```
Введите количество усредняемых чисел: 5

Введите число типа double №1: 1
Введите число типа double №2: 2
Введите число типа double №3: 3
Введите число типа double №4: 4
Введите число типа double №5: 5

3 является средним из (1 + 2 + 3 + 4 + 5) / 5
Нажмите <Enter> для завершения программы...
```

Сначала программа `VariableArrayAverage` выводит приглашение пользователю указать количество значений, которые будут введены далее и которые нужно усреднить. Введенное значение сохраняется в переменной `numElements` типа `int`. В представленном примере введено число 5.

Затем программа выделяет память для нового массива `doublesArray` с указанным количеством элементов. В данном случае она делает это для массива, состоящего из пяти элементов типа `double`. Программа выполняет `numElements` итераций цикла, считывая вводимые пользователем значения и заполняя ими массив. После того как пользователь введет указанное им ранее число данных, программа использует тот же алгоритм, что и в программе `FixedArrayAverage` для вычисления среднего значения последовательности чисел. В последней части генерируется вывод среднего значения вместе с введенными числами в привлекательном виде.


СОВЕТ

Этот вывод не так уж и прост, как может показаться. Внимательно проследите, как именно программа выводит открывающую скобку, знаки сложения, числа последовательности и закрывающую скобку.

Программа `VariableArrayAverage`, возможно, не удовлетворяет вашим представлениям о гибкости. Возможно, вы бы хотели позволить пользователю вводить числа, а после ввода какого-то очередного числа дать команду вычислить среднее значение введенных чисел. Кроме массивов, C# предоставляет программисту и другие типы коллекций; некоторые из них могут при необходимости увеличивать или уменьшать свой размер.

Свойство Length

В программе VariableArrayAverage для заполнения массива использован цикл for:

```
// Объявляем массив необходимого размера
double[] doublesArray = new double[numElements];

// Накапливаем значения в массиве
for (int i = 0; i < numElements; i++)
{
    . . .
}
```

Массив doublesArray объявлен как имеющий длину numElements. Таким образом, понятно, почему цикл выполняет именно numElements итераций для прохода по массиву.

Вообще говоря, не слишком-то удобно таскать повсюду вместе с массивом переменную, в которой хранится его длина. Но, к счастью, это не является неизбежным — у массива есть свойство Length, которое содержит его длину, так что doublesArray.Length в данном случае имеет то же значение, что и numElements.

Таким образом, предпочтительнее использовать такой вид цикла for:

```
// Накапливаем значения в массиве
for (int i = 0; i < doublesArray.Length; i++)
```

Инициализация массивов

С первого взгляда бросается в глаза, насколько различаются синтаксисы массивов фиксированной и переменной длины:

```
double[] initializedArray = {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
double[] blankArray = new double[10];
```

ЗАПОМНИ!

Несмотря на то что blankArray выделяет место для элементов, по-прежнему необходимо инициализировать его значения. Для выполнения этой задачи путем присвоения значения каждому индексированному элементу можно, например, использовать цикл for.

Цикл foreach

Рассмотрим пример, в котором вычисляется средняя успеваемость студентов:

```
public class Student  // С классами вы познакомитесь позже
{
  public string name;
  public double gpa; // Средний балл
}
```

```
public class Program
{
  public static void Main(string[] args)
  {
    // ... Создаем массив ...

    // Усредняем успеваемость
    double sum = 0.0;
    for (int i = 0; i < students.Length; i++)
    {
      sum += students[i].gpa;
    }
    double avg = sum / students.Length;
    // ... Прочие действия с массивом ...
  }
}
```

Цикл `for` проходит по всем элементам массива. (Массив может содержать не только простые величины типа `int` или `double`, но и объекты классов. Формально вы пока что не знакомы с классами, но это не важно.)

Переменная `students.Length` содержит количество элементов в массиве.

ЗАПОМНИ!

Язык C# предоставляет программистам особую конструкцию цикла, `foreach`, которая спроектирована специально для итеративного прохода по контейнерам, таким как массивы. Она работает следующим образом:

```
// Усредняем успеваемость
double sum = 0.0;
foreach (Student student in students)
{
    sum += student.gpa;
}
double avg = sum / students.Length;
```

При первом входе в цикл из массива выбирается первый объект типа `Student` и сохраняется в переменной `student`. При каждой последующей итерации цикл `foreach` выбирает из цикла и присваивает переменной `student` очередной элемент массива. Управление покидает цикл `foreach`, когда все элементы массива оказываются обработанными.

Обратите внимание на то, что в выражении `foreach` нет никаких индексов. Это позволяет существенно снизить вероятность появления ошибки в программе.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

На самом деле цикл `foreach` мощнее, чем можно представить из приведенного примера. Он работает не только с массивами, но и с другими видами коллекций. Кроме того, `foreach` может работать и с многомерными массивами (т.е. массивами массивов), но эта тема выходит за рамки настоящей книги. Поищите тему *Multidimensional arrays* в справочной системе.

Сортировка массива данных

Сортировка элементов в массиве — весьма распространенная программистская задача. То, что массив не может увеличиваться или уменьшаться, еще не означает, что его элементы не могут перемещаться, удаляться или добавляться. Например, обмен (взаимная перестановка) двух элементов типа `string` в массиве может быть выполнен так, как показано в следующем фрагменте исходного текста:

```
string temp = strings[i]; // Сохраняем i-ю строку
strings[i]  = strings[k]; // Заменяем i-ю строку k-й
strings[k]  = temp;       // Заменяем k-ю строку temp
```

Здесь сначала во временной переменной сохраняется ссылка на объект в *i*-й позиции массива `strings`, чтобы она не была потеряна при обмене, затем ссылка в *i*-й позиции заменяется ссылкой в *k*-й позиции. После этого в *k*-ю позицию помещается ранее сохраненная во временной переменной ссылка, которая изначально находилась в *i*-й позиции. Происходящее схематично показано на рис. 6.1.

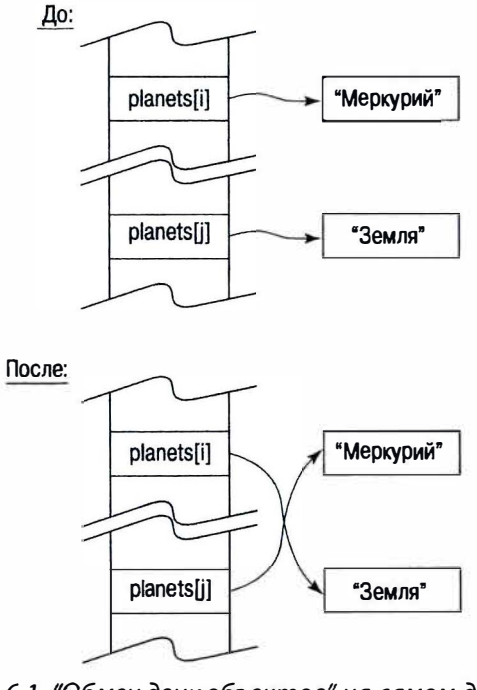


Рис. 6.1. "Обмен двух объектов" на самом деле означает "обмен ссылок на два объекта"

СОВЕТ

Некоторые коллекции данных более гибки, чем массивы, и поддерживают добавление и удаление элементов. В приведенной ниже программе демонстрируется, как использовать возможность манипуляции элементами массива для их сортировки. В программе применен алгоритм *пузырьковой сортировки*. Он не слишком эффективен и плохо подходит для сортировки больших массивов с тысячами элементов, но зато очень прост и вполне приемлем для небольших массивов.

```
// BubbleSortArray - сортирует список планет по именам:
//    1. В алфавитном порядке
//    2. По длине имен от коротких к длинным
//    3. По длине имен от длинных к коротким
// Использованы два алгоритма сортировки:
//    1. Алгоритм Sort использован методом Sort()
//    2. Классический алгоритм пузырьковой сортировки
using System;
namespace BubbleSortArray
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("5 ближайших к Солнцу планет: ");
      string[] planets = new string[]
        { "Mercury", "Venus", "Earth", "Mars", "Jupiter" }
        //  { "Меркурий", "Венера", "Земля",
        //    "Марс", "Юпитер" }
      foreach (string planet in planets)
      {
        // Символ \t вставляет табуляцию при выводе
        Console.WriteLine("\t" + planet);
      }

      Console.WriteLine("\nВ алфавитном порядке: ");
      // Array.Sort() - метод класса Array
      // Array.Sort() работает в пределах массива, не
      // оставляя исходной копии. Решение состоит в
      // копировании старого массива и работе с копией
      string[] sortedNames = planets;
      Array.Sort(sortedNames);

      // Показываем, что sortedNames содержит те же
      // планеты, но отсортированные
      foreach (string planet in sortedNames)
      {
        Console.WriteLine("\t" + planet);
      }

      Console.WriteLine("\nСортировка по длине имени: ");
      // Алгоритм пузырьковой сортировки — самый простой и
      // неэффективный. Метод Array.Sort() существенно
```

```
            // эффективнее, но здесь он неприемлем, так как
            // сравниваются не строки, а их длины
            int outer; // Индекс внешнего цикла
            int inner; // Индекс внутреннего цикла
            // Цикл от последнего индекса к первому
            for (outer = planets.Length - 1; outer >= 0; outer--)
            {
              // На каждом цикле проходим по всем элементам
              // ниже текущего. Этот цикл проходит в восходящем
              // порядке. Цикл for позволяет обход в любом
              // направлении
              for (inner = 1; inner <= outer; inner++)
              {
                // Сравниваем соседние элементы. Если ранний более
                // длинный, обмениваем их местами
                if (planets[inner - 1].Length >
                    planets[inner].Length)
                {
                  string temp = planets[inner - 1];
                  planets[inner - 1] = planets[inner];
                  planets[inner] = temp;
                }
              }
            }

            foreach (string planet in planets)
            {
              Console.WriteLine("\t" + planet);
            }

            Console.WriteLine("\nВ обратном порядке: ");
            // Цикл в обратном порядке
            for(int i = planets.Length - 1; i >= 0; i--)
            {
              Console.WriteLine("\t" + planets[i]);
            }

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
          }
        }
      }
```

Программа начинается с массива, содержащего имена первых пяти планет, ближайших к Солнцу. Затем программа вызывает метод `Sort()` массива. После сортировки встроенным методом `Sort()` класса `Array` программа сортирует имена по их длинам с помощью пользовательского алгоритма сортировки.

Встроенный метод массивов и коллекций `Sort()` намного эффективнее пузырьковой сортировки. Не надо использовать собственные подпрограммы, если

у вас нет на то веских причин. Алгоритм второй сортировки работает путем постоянных обходов списка, пока он не будет отсортирован. При каждом проходе массива `sortedNames` программа сравнивает каждую строку с соседней. Если две строки расположены в неверном порядке, алгоритм меняет их местами и выставляет флаг, указывающий, что список не был полностью отсортирован. На рис. 6.2–6.5 показано состояние списка `planets` после очередного прохода.

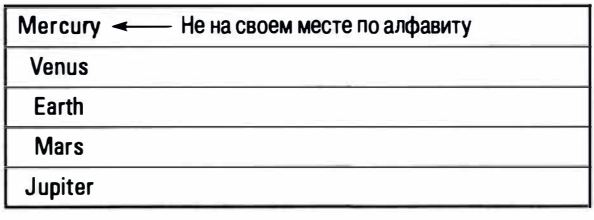


Рис. 6.2. До начала пузырьковой сортировки

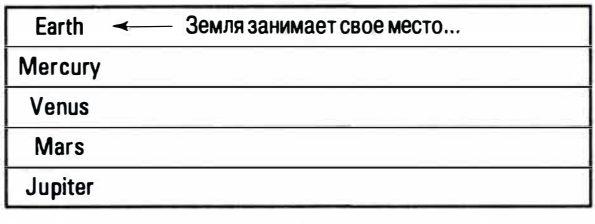


Рис. 6.3. После первого прохода пузырьковой сортировки

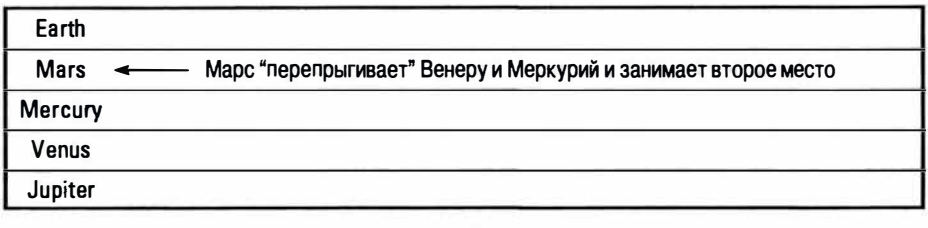


Рис. 6.4. После второго прохода пузырьковой сортировки

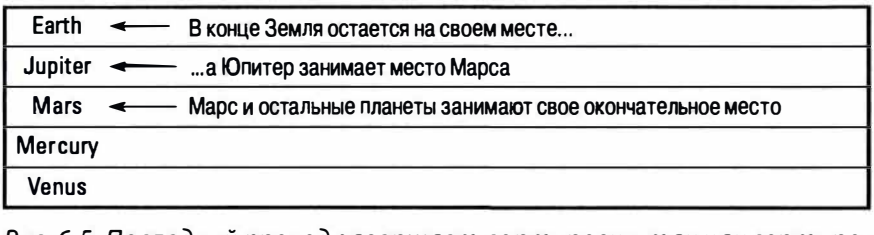


Рис. 6.5. Последний проход завершает сортировку, так как сортировать больше нечего

При последней сортировке по длинам имен короткие имена "всплывают" к вершине списка (вот откуда произошло название *пузырьковая сортировка*).

Обратите внимание, что в коде единичным переменным даются имена в единственном числе, такие как `planet` или `student`. Лучше, если имя переменной каким-то образом включает имя класса, например `badStudent` или `goodStudent`. Массивам (или иным коллекциям) лучше давать имена во множественном числе, например `students` или `phoneNumbers`.

Использование `var` для массивов

Традиционно используется один из следующих видов инициализации массивов (эти виды имеют тот же возраст, что и сам C#):

```
int[] numbers =
    new int[3];                 // Размер без инициализации
int[] numbers =
    new int[] { 1, 2, 3 };      // Инициализация без размера
int[] numbers =
    new int[3] { 1, 2, 3 };     // Размер и инициализация
int[] numbers = { 1, 2, 3 };    // Краткая форма без 'new'
```

В главе 2, "Работа с переменными", было введено ключевое слово `var`, которое говорит компилятору C#: "Выведи тип данных переменной из инициализирующего выражения, которое я предоставил, самостоятельно". К счастью, ключевое слово `var` работает и с массивами:

```
// myArray  -  массив int[] с 6 элементами
var myArray =                          // Инициализация
    new [] { 2, 3, 5, 7, 11, 13 }; // обязательна!
```

Новый синтаксис имеет только два изменения:

- использование ключевого слова `var` вместо явного указания типа данных массива `numbers` в левой части присваивания;
- пропуск ключевого слова `int` перед скобками в правой части присваивания (часть, которая выводится компилятором).

ЗАПОМНИ!

Обратите внимание на необходимость наличия инициализатора в версии с ключевым словом `var`. Компилятор использует его для выведения типа элементов массива без ключевого слова `int`. Вот еще несколько примеров:

```
var names = new [] { "John", "Paul",
                     "George", "Ringo" }; // string
var averages = new [] { 3.0, 3.34,
                        4.0, 2.0, 1.8 };  // double
var prez = new []{ new President("FDR"),
                   new President("JFK")}; // President
```

ЗАПОМНИ!

Обратите внимание на то, что при использовании ключевого слова `var` краткая форма без `new` употребляться не может. Приведенный далее код компилироваться не будет:

```
var names = { "John", "Paul", "George", "Ringo" }; // Нет 'new []'
```

Способ с использованием `var` более многословен, но в некоторых ситуациях он весьма неплох, а иногда и просто необходим (с такими ситуациями вы встретитесь в главе 7, "Работа с коллекциями").

Коллекции C#

Использование массивов — простейший способ работы со списками студентов или чисел с плавающей точкой. В .NET имеется множество мест, где необходимо применять массивы. Однако массивы имеют ряд серьезных ограничений, которые могут полностью исключить их использование в некоторых приложениях. В таких ситуациях можно подумать о применении более гибких классов коллекций C#. Хотя массивы просты в использовании и могут иметь несколько измерений, их слабым местом являются следующие важные ограничения.

- Программа должна объявить размер массива при его создании. В отличие от Visual Basic, C# не позволяет изменять размер массива после того, как он определен. Но что если заранее неизвестно, какой размер должен иметь массив?
- Вставка или удаление элемента в средине массива очень неэффективна: приходится перемещать все элементы для освобождения места. В случае большого массива это может существенно снизить производительность приложения.

В большинстве коллекций добавление, вставка или удаление элементов выполняется существенно проще; кроме того, можно изменять их размер в процессе работы. В действительности изменение размера при необходимости выполняется автоматически.

СОВЕТ

Если вам нужна многомерная структура данных, используйте массив. Коллекции не позволяют работать с несколькими измерениями (хотя можно создавать такие сложные структуры данных, как коллекции массивов или коллекций). У массивов и коллекций много общего.

- Они могут содержать элементы одного и только одного типа. Вы должны указать этот тип в своем коде, чтобы он был известен во время компиляции. Объявленный тип нельзя изменять (но прочтите последний раздел данной главы).

» Как и в случае массивов, к элементам большинства коллекций можно обращаться при помощи синтаксиса обращения к элементам массива с использованием индекса: `myList[3] = "Joe"`.

» И коллекции, и массивы имеют методы и свойства. Таким образом, чтобы найти количество элементов в массиве `smallPrimeNumbers`, используйте свойство `Length`:

```
var smallPrimeNumbers = new [] { 2, 3, 5, 7, 11, 13 };
int numElements = smallPrimeNumbers.Length; // Результат - 6
```

Однако в случае коллекции соответствующее свойство называется `Count`:

```
List<int> smallPrimes = new List<int> { 2, 3, 5, 7, 11, 13 };
int numElements = smallPrimes.Count;
```

» Какие именно методы и свойства имеет класс `Array`, можно узнать в справочной системе (7 открытых свойств и 36 открытых методов).

Синтаксис коллекций

В этом разделе вы познакомитесь с синтаксисом коллекций и наиболее важными и часто используемыми классами коллекций. В табл. 6.1 перечислены основные классы коллекций C#. Их полезно представлять как имеющие различные "формы" — коллекция в виде списка или, например, словаря.

Таблица 6.1. Наиболее распространенные коллекции

Класс	Описание
`List<T>`	Динамический массив, содержащий объекты типа `T`
`LinkedList<T>`	Связанный список объектов типа `T`
`Queue<T>`	Список "первым вошел, первым вышел"
`Stack<T>`	Список "последним вошел, первым вышел"
`Dictionary<TKey, TValue>`	Структура, работающая как словарь. Коллекция пар "ключ–значение", в которой возвращается значение, соответствующее заданному ключу
`HashSet<T>`	Новая структура, аналогичная математическому множеству без повторяющихся элементов. Очень похожа на список, но предоставляет такие математические операции, как объединение и пересечение множеств

Понятие <T>

В этих странно выглядящих записях в табл. 6.1 <T> обозначает место, куда будет помещен некоторый реальный тип. Чтобы вызвать к жизни этот символический объект, его *инстанцируют* путем указания реального типа:

```
List<int> intList = new List<int>(); // Инстанцирование для int
```


ЗАПОМНИ!

Инстанцирование означает создание объекта (экземпляра) определенного типа. Например, можно инстанцировать `List<T>` для типов `int`, `string`, `Student` и т.д. Кстати говоря, `T` — отнюдь не священная корова, и вместо него можно использовать все что угодно, например `<dummy>` или `<myType>`. Обычно для параметра типа применяются буквы `T`, `U`, `V` и т.д. Обратите внимание на коллекцию `Dictionary<TKey, TValue>` в табл. 6.1. Здесь требуются два типа — для ключа и для значений, связанных с ключами. Немного позже вы узнаете, как пользоваться таким словарем.

Обобщенные коллекции

Эти современные коллекции известны как *обобщенные* (generic). Они являются обобщенными в том смысле, что пустой шаблон можно заполнить любым типом для создания пользовательской коллекции. Если обобщенный список `List<T>` кажется вам какой-то головоломкой, вам поможет разобраться в ней глава 8, "Обобщенность".

Использование списков

Предположим, нужно сохранить список объектов `MP3`, каждый из которых представляет один элемент вашей MP3-коллекции. При использовании массива это может выглядеть следующим образом:

```
MP3[] myMP3s = new MP3[50]; // Начнем с пустого массива
// Создаем MP3 и добавляем в массив:
myPP3s[0] = new MP3("Norah Jones");
// ... и так далее
```

В случае списка код будет выглядеть следующим образом:

```
List<MP3> myMP3s = new List<MP3>(); // Пустой список
// Вызываем метод Add() для добавления в список
myMP3s.Add(new MP3("Avril Levigne"));
// ... и так далее
```

Код выглядит почти одинаково, и никаких преимуществ списка над массивом не видно. Но что произойдет после того, как вы добавите пятьдесят

объектов MP3 в массив и захотите добавить пятьдесят первый? Для него не найдется места. Вам придется объявлять новый, больший массив, а затем копировать в него все объекты из старого. При удалении объектов из массива в нем будут образовываться пустые места. Что вы будете помещать в эти пустые ячейки массива? Возможно, значение null?

Список легко решает все описанные проблемы. Добавить MP3 номер 51? Без проблем. Удалить объект из списка? Нет вопросов. Список сам побеспокоится о своем состоянии после удаления.


ВНИМАНИЕ!

Если ваш список (или массив) может содержать объекты null, при использовании цикла for или foreach следует проверять объекты на равенство null. Вы не должны вызывать метод Play() для объекта null — это обязательно приведет к ошибке.

Инстанцирование пустого списка

В следующем фрагменте показано инстанцирование нового, пустого списка объектов типа string. Другими словами, список предназначен для хранения только строк:

```
// List<T>: обратите внимание на угловые скобки в объявлении
// List<T> ; T представляет собой "параметр типа",
// List<T> – "параметризованный тип".
// Инстанцирование для типа string.
List<string> nameList = new List<string>();
sList.Add("one");
sList.Add(3);                          // Ошибка компиляции!
sList.Add(new Student("du Bois")); // Ошибка компиляции!
```

Элементы в список List<T> добавляются при помощи метода Add(). Приведенный выше код успешно добавляет строку в список, но при попытке добавить целое число или объект типа Student компилятор сообщает об ошибке. Данный список инстанцирован для строк, так что обе попытки добавить не строку отвергаются компилятором.

Создание списка целых чисел

В следующем фрагменте кода инстанцируется новый список для типа int, затем в него добавляются два значения int. После этого цикл foreach проходит по списку, выводя хранящиеся в нем числа.

```
// Инстанцирование для int
List<int> intList = new List<int>();
intList.Add(3);        // Все в порядке
intList.Add(4);
Console.WriteLine("Вывод списка:");
// foreach работает для любой коллекции:
```

```
foreach(int i in intList)
{
    Console.WriteLine("int i = " + i);
}
```

Создание списка для хранения объектов

В очередном фрагменте кода инстанцируется новый список для хранения объектов `Student` и в него добавляются два объекта при помощи метода `Add()`. Обратите внимание, как затем в список вносится *массив* объектов `Student`, для чего используется метод `AddRange()`. Этот метод позволяет добавить в список массив или практически любую коллекцию одним действием:

```
// Инстанцирование для типа Student.
List<Student> studentList = new List<Student>();
Student student1 = new Student("Vigil");
Student student2 = new Student("Finch");
studentList.Add(student1);
studentList.Add(student2);
Student[] students =
    { new Student("Mox"), new Student("Fox") };
studentList.AddRange(students); // Добавление массива
Console.WriteLine("Число объектов в studentList = " +
                  studentList.Count);
```

`List<T>` имеет также ряд других методов для добавления элементов, включая методы для вставки одного или нескольких элементов в произвольное место списка, а также методы для удаления элементов и очистки списка.

Преобразования списков в массивы и обратно

Список и массив легко преобразовать один в другой. Чтобы внести массив в список, используется метод списка `AddRange()`, показанный выше. Для преобразования списка в массив используется метод списка `ToArray()`:

```
Student[] students =       // studentList представляет
    studentList.ToArray(); // собой List<Student>
```

Подсчет количества элементов в списке

Для определения количества элементов в `List<T>` используется свойство `Count`. Это может приводить к неприятностям, если вы привыкли к свойству `Length` у массивов и строк. Запомните — у коллекций это свойство называется `Count`.

Поиск в списках

Есть несколько способов поиска элемента в списке. Метод `IndexOf()` возвращает индекс искомого элемента в списке (`-1`, если таковой элемент в списке отсутствует). В приведенном далее коде демонстрируется также обращение к элементу с помощью индексов и метода `Contains()`. Другие методы поиска элементов, включая метод `BinarySearch()`, здесь не показаны.

```
// Поиск с использованием IndexOf().
Console.WriteLine("Индекс Student2 - " +
                  studentList.IndexOf(student2));
string name = studentList[3].Name; // Доступ по индексу.
if(studentList.Contains(student1)) // student1 - объект
{                                  // типа Student.
    Console.WriteLine(student1.Name + " contained in list");
}
```

Прочие действия со списками

В приведенном ниже фрагменте представлено еще несколько операций со списком `List<T>`, включая сортировку, вставку и удаление элементов:

```
// Предполагается, что класс Student реализует
// интерфейс IComparable
studentList.Sort();
studentList.Insert(3, new Student("Ross"));
studentList.RemoveAt(3);    // Удаление третьего элемента
Console.WriteLine("removed " +
                  name);    // name определено выше
```

Это лишь часть методов `List<T>`. Полный список методов можно найти в справочной системе. Чтобы найти обобщенные коллекции в справочной системе, в ее указателе следует искать *List<T>*. Если вы введете просто *List*, то потеряетесь в списке списков списков... Если вас интересует информация обо всем множестве классов обобщенных коллекций, поищите в предметном указателе *generic collections*.

Использование словарей

Вы наверняка работали с разными словарями. Словарь представляет собой набор слов в алфавитном порядке, и с каждым словом связана некоторая информация, включающая пояснения, определения и т.д. При использовании словаря вы просто ищете интересующее вас слово и получаете относящуюся к нему информацию.

Словарь в C# существенно отличается от списка: он представлен классом `Dictionary<TKey, TValue>` с двумя параметрами типа. `TKey` представляет тип данных, используемый в качестве *ключа* словаря (как обычном словаре — слова, по которым выполняется поиск). `TValue` представляет тип данных, используемый для хранения информации, связанной с ключом, как пояснения в толковом словаре. Далее рассказывается, как работать со словарем.

Создание словаря

Первый фрагмент кода создает новый объект `Dictionary` и ключ, значения которого имеют тип `string`. Само собой, вы не ограничены строками! Как ключ, так и значение могут быть любого типа. Обратите внимание на то, что метод `Add()` требует передачи и ключа, и значения.

```
Dictionary<string, string> dict = new Dictionary<string, string>();
// Add(key, value)
dict.Add("C#",      "Круто");
dict.Add("C++",     "Стихи на санскрите азбукой Морзе");
dict.Add("VB",      "Просто, но многословно");
dict.Add("Java",    "Неплохо, но не C#");
dict.Add("Fortran", "ДРВНСТ"); // Переменная максимально допустимой
                               // в Fortran длины (6 символов),
                               // означающая "Древность"
dict.Add("Cobol",   "Еще более многословно, чем VB");
```

Поиск в словаре

Метод `ContainsKey()` позволяет узнать, имеется ли в словаре определенный ключ. Имеется также соответствующий метод `ContainsValue()`.

```
// Есть ли в словаре определенный ключ
Console.WriteLine("Содержит ключ C# " +
                  dict.ContainsKey("C#"));   // True
Console.WriteLine("Содержит ключ Ruby " +
                  dict.ContainsKey("Ruby")); // False
```

Пары в словаре не находятся в каком-то определенном порядке, и вы не можете сортировать словарь. Он напоминает кучу коробок с информацией, разбросанных по всей комнате.

Итерирование словаря

Конечно же, словарь, как и любую другую коллекцию, можно итерировать. Но при этом надо помнить, что словарь скорее напоминает список *пар* — объектов, которые содержат и ключ, и значение. Поэтому при полном обходе словаря при помощи цикла `foreach` на каждой итерации выбирается одна из пар, которые представляют собой объекты типа `KeyValuePair<TKey, TValue>`.

В вызове `WriteLine()` я воспользовался свойствами пары `Key` и `Value` для получения соответствующих элементов. Вот как все это выглядит:

```
// Обход словаря при помощи цикла foreach
// Выбираются пары "ключ-значение"
Console.WriteLine("\nСодержимое словаря:");
foreach (KeyValuePair<string, string> pair in dict)
{
    // Так как ключ представляет собой строку,
    // я могу вызывать соответствующие методы
    Console.WriteLine("Ключ: " + pair.Key.PadRight(8) +
                      "Значение: " + pair.Value);
}
```

В следующем фрагменте кода показано, как можно итерировать только ключи или только значения словаря. Свойство словаря `Keys` возвращает другую коллекцию, а именно — списочную коллекцию типа `Dictionary<TKey,TValue>.KeyCollection`. Поскольку наши ключи представляют собой строки, можно итерировать ключи как строки и вызывать для них соответствующие методы. Свойство `Values` аналогично свойству `Keys`. В последней строке фрагмента используется свойство `Count`, которое позволяет узнать, сколько всего пар "ключ–значение" имеется в словаре.

```
// Перечисляем ключи (без упорядочения)
Console.WriteLine("\n>>>>> Ключи:");

// Dictionary<TKey, TValue>.KeyCollection - коллекция
// ключей, в данном случае — строк
Dictionary<string, string>.KeyCollection keys = dict.Keys;
foreach(string key in keys)
{
    Console.WriteLine("Ключ: " + key);
}

// Перечисляем значения, находящиеся в
// том же порядке, что и ключи
Console.WriteLine("\n>>>>> Значения:");
Dictionary<string, string>.ValueCollection values = dict.Values;
foreach (string value in values)
{
    Console.WriteLine("Значение: " + value);
}
Console.Write("\nКоличество элементов в словаре: " + dict.Count);
```

Конечно, возможности словарей этим не исчерпываются. Обратитесь к разделу *generic dictionary* справочной системы за полной информацией.

Инициализаторы массивов и коллекций

В этом разделе речь пойдет о методах инициализации массивов и коллекций — в старом и в новом стилях. Можете загнуть уголок этой страницы, чтобы потом к ней вернуться.

Инициализация массивов

Напомню, что синтаксис `var`, рассматривавшийся в этой главе ранее, позволяет объявлять массивы следующим образом:

```
int[] numbers = {1, 2, 3};      // Краткая форма -
                                // без использования var
var numbers = new []{1, 2, 3}; // При использовании var
                                // требуется полный инициализатор
```

Инициализация коллекций

Традиционный способ инициализации коллекции, такой как `List<T>` (или `Queue<T>` или `Stack<T>`), во времена C# 2.0 выглядел следующим образом:

```
List<int> numList = new List<int>(); // Пустой список
numbers.Add(1);                      // Добавление элементов
numbers.Add(2);                      // По одному...
numbers.Add(3);                      // ...доооооолго!
```

Если у вас есть эти числа в другой коллекции или массиве, можно поступить немного по-другому:

```
List<int> numList = new List<int>(numbers); // Из массива
List<int> numList2 = new List<int>(numList);// Из коллекции
numList.AddRange(numbers);     // При помощи вызова AddRange
```

Начиная с C# 3.0 инициализаторы коллекций напоминают инициализаторы массивов и существенно проще в использовании, чем более ранние способы. Новые инициализаторы имеют следующий вид:

```
List<int> numList = new List<int> { 1, 2, 3 }; // Список
int[] intArray = { 1, 2, 3 };                  // Массив
```

Ключевое различие между новыми инициализаторами массивов и коллекций состоит в том, что для коллекций обязательно нужно указывать тип данных, т.е. после ключевого слова `new` следует указывать `List<int>`.

Вы можете использовать новое ключевое слово `var` и с коллекциями:

```
var list = new List<string> { "Head", "Heart",
                              "Hands", "Health" };
```

Можно также использовать новое ключевое слово dynamic:

```
dynamic list = new List<string>{"Head","Heart","Hands","Health"};
```

Инициализация словарей с использованием нового синтаксиса почти такая же:

```
Dictionary<int, string> dict =
    new Dictionary<int, string> { { 1, "Sam" },
                                  { 2, "Joe" } };
```

Внешне все выглядит так же, как и для List<T>, но внутри внешних фигурных скобок имеется второй уровень скобок, по паре скобок для каждого элемента словаря. Поскольку данный словарь dict имеет целые ключи и строковые значения, каждая внутренняя пара фигурных скобок содержит число и строку, разделенные запятыми. Пары "ключ–значение" также разделены запятыми.

Инициализация множеств (о которых я расскажу в следующем разделе) очень похожа на инициализацию списков:

```
HashSet<int> biggerPrimes =
    new HashSet<int> { 19, 23, 29, 31, 37, 41 };
```

Использование множеств

В C# 3.0 добавлен новый тип коллекции — HashSet<T>. *Множество* (set) представляет собой неупорядоченную коллекцию неповторяющихся элементов. Концепция множества происходит из математики. В качестве примеров множеств можно привести множество дней недели, учеников в классе, целых чисел и т.п. В отличие от математических множеств множества C# не могут быть бесконечными, но их размер ограничен только доступной памятью. В последующих разделах будет рассказано, как работать с множествами в своих программах.

Выполнение специфичных для множеств задач

Так же, как и в случае с другими коллекциями, вы можете добавлять элементы в множество, удалять их или искать. Но можно выполнять и некоторые специфичные для множества операции, такие как *объединение* и *пересечение* множеств[1].

[1] Об операциях над множествами можно прочесть в Интернете, например, по адресу https://www.probabilitycourse.com/chapter1/1_2_2_set_operations.php.

- **Объединение**: сливает элементы двух множеств в одно.
- **Пересечение**: находит элементы, которые имеются одновременно в обоих множествах, и возвращает новое множество, состоящее только из этих элементов.
- **Разность**: определяет, какие элементы одного множества отсутствуют в другом.

Когда следует использовать `HashSet<T>`? В любое время, когда вы работаете с двумя или более коллекциями и хотите найти, например, их пересечение (или создать коллекцию, которая содержит две другие коллекции, или исключить группу элементов из коллекции). Многие методы `HashSet<T>` могут связывать множества и другие классы коллекций. Конечно, множества способны на большее, так что поищите термин `HashSet<T>` в справочной системе по языку программирования C#.

Создание множества

Для создания объекта типа `HashSet<T>` можно выполнить следующие действия:

```
HashSet<int> smallPrimeNumbers = new HashSet<int>();
smallPrimeNumbers.Add(2);
smallPrimeNumbers.Add(3);
```

Более удобно воспользоваться инициализатором коллекции:

```
HashSet<int> smallPrimeNumbers =
    new HashSet<int> { 2, 3, 5, 7, 11, 13 };
```

Можно также создать множество из существующей коллекции наподобие списка или из массива:

```
List<int> intList = new List<int> {0, 1, 2, 3, 4, 5, 6, 7};
HashSet<int> numbers = new HashSet<int>(intList);
```

Добавление элемента в множество

Если вы попытаетесь добавить элемент в множество, в котором этот элемент уже имеется (`smallPrimeNumbers.Add(2);`), то, с одной стороны, это не будет воспринято как ошибка, но с другой — само множество при этом не изменится (так как в множестве не может быть дубликатов). Метод `Add()` вернет `true`, если добавление выполнено, и `false` — в противном случае. Вы не обязаны проверять возвращаемое значение, но оно может оказаться полезным, если вы захотите выполнить какие-то действия в случае добавления в множество дубликата уже содержащегося в нем элемента:

```
bool successful = smallPrimeNumbers.Add(2);
if(successful)
{
    // 2 добавлено, можно выполнить какие-то
    // связанные с этим действия
}
```

Выполнение объединения

В приведенном далее примере демонстрируется ряд методов `HashSet<T>`, но, что более важно, в нем показано использование `HashSet<T>` *в качестве инструмента для работы с другими коллекциями*. `HashSet<T>` позволяет выполнять строгие математические операции, но его возможность комбинировать коллекции различными способами представляется мне особенно удобной.

Первый фрагмент этого кода начинается со списка `List<string>` и массива. Каждый из них содержит названия цветов. Если вы объедините их при помощи простого вызова метода списка `AddRange()` (например, `colors.AddRange(moreColors);`), то полученный в результате список будет иметь дубликаты (`yellow`, `orange`). Используя же `HashSet<T>` и метод `UnionWith()`, можно объединить две коллекции и удалить все дубликаты одним действием, как показано в следующем примере.

```
Console.WriteLine("Объединение коллекций без дубликатов:");
List<string> colors =
    new List<string> { "red", "orange", "yellow" };
string[] moreColors =
    { "orange", "yellow", "green", "blue", "violet" };
// Первая стадия - создание множества...
HashSet<string> combined = new HashSet<string>(colors);
// ...вторая стадия - объединение элементов из
// обоих списков без дубликатов
combined.UnionWith(moreColors);
foreach (string color in combined)
{
    Console.WriteLine(color);
}
```

В результате мы получаем множество, которое содержит элементы `"red"`, `"orange"`, `"yellow"`, `"green"`, `"blue"` и `"violet"`. На первом этапе список `colors` используется для инициализации нового множества `HashSet<T>`. На втором этапе вызывается метод множества `UnionWith()`, который добавляет к множеству массив `moreColors`, но фактически добавляет только те элементы массива, которых еще нет в множестве. В конце концов множество содержит все цвета, которые имеются в обоих исходных множествах.

Но предположим, что вам нужен список List<T> с этими цветами, а не множество HashSet<T>. В приведенном ниже фрагменте показано, как создать новый список List<T>, инициализированный множеством combined:

```
Console.WriteLine("\nПреобразуем множество в список:");
// Новый список инициализируется массивом
List<string> spectrum = new List<string>(combined);
foreach(string color in spectrum)
{
    Console.WriteLine(color);
}
```

Пересечение множеств

Представьте, что вам нужно работать в предвыборной кампании президента США с примерно десятком исходных кандидатов от каждой из основных партий. Немало из них являются сенаторами. Как получить список кандидатов-сенаторов? Метод IntersectWith() класса HashSet<T> позволяет получить элементы, присутствующие одновременно в списке кандидатов в президенты и в списке сенаторов:

```
Console.WriteLine("\nПоиск перекрытия двух списков:");
List<string> presidentialCandidates =
    new List<string> { "Clinton", "Edwards", "Giuliani",
                       "McCain", "Obama", "Romney" };
List<string> senators =
    new List<string> { "Alexander", "Boxer", "Clinton",
                       "McCain", "Obama", "Snowe" };
HashSet<string> senatorsRunning =
    new HashSet<string>(presidentialCandidates);
// IntersectWith() находит те элементы, которые имеются
// в обоих множествах, удаляя остальные
senatorsRunning.IntersectWith(senators);
foreach (string senator in senatorsRunning)
{
    Console.WriteLine(senator);
}
```

В результате мы получим "Clinton", "McCain" и "Obama", поскольку только они присутствуют в обоих списках.

В приведенном далее фрагменте кода используется метод SymmetricExceptWith(), который дает результат, противоположный методу IntersectWith(). В то время как пересечение дает только те элементы, которые присутствуют в обоих списках, SymmetricExceptWith() дает те элементы из обоих списков, которые *присутствуют только в одном из них*. В итоге мы получаем множество из элементов 5, 3, 1, 12 и 10:

```
Console.WriteLine("\nНеперекрывающиеся элементы списков:");
Stack<int> stackOne =
    new Stack<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8 });
Stack<int> stackTwo =
    new Stack<int>(new int[] { 2, 4, 6, 7, 8, 10, 12 });
HashSet<int> nonoverlapping = new HashSet<int>(stackOne);
// SymmetricExceptWith() собирает элементы, которые есть
// только в одной из коллекций, но не в обеих одновременно
nonoverlapping.SymmetricExceptWith(stackTwo);
foreach(int n in nonoverlapping)
{
    Console.WriteLine(n.ToString());
}
```

Использование стеков здесь немного нестандартное, потому что код добавляет все элементы в стек одновременно, а не вносит каждый из них по отдельности. Корректные операции со стеком заключаются во внесении элементов в стек по одному и в таком же их снятии со стека.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Обратите внимание на то, что все продемонстрированные методы `HashSet<T>` имеют тип `void`, т.е. они не возвращают значения. Таким образом, результат непосредственно отображается в множестве, для которого вызван метод; в приведенном выше фрагменте это множество `nonoverlapping`.

Получение разности

Противоположная задача заключается в том, чтобы удалить любые элементы, которые имеются в обоих списках, так, чтобы в конечном итоге получить в результирующем списке только те элементы, которые отсутствуют в другом списке. Это выполняется с помощью метода `ExceptWith()` класса `HashSet<T>`:

```
Console.WriteLine("\nИсключение элементов из списка:");
Queue<int> queue =
    new Queue<int>(new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 17 });
HashSet<int> unique =
    new HashSet<int> { 1, 3, 5, 7, 9, 11, 13, 15 };
// ExceptWith() удаляет из unique элементы, которые
// имеются в наличии в queue: 1, 3, 5, 7.
unique.ExceptWith(queue);
foreach (int n in unique)
{
    Console.WriteLine(n.ToString());
}
```

После этого кода из `unique` оказываются исключены его элементы, которые имеются в `queue` (1, 3, 5, 7 и 9) и в конечном итоге в `unique` остаются только элементы 11, 13 и 15.

Не используйте старые коллекции

Изначально классы коллекций были реализованы как коллекции для типа `Object`, так что было невозможно создать коллекцию конкретно для `string` или для `int`. Такая коллекция позволяла хранить любой тип данных, поскольку все объекты в C# порождены от класса `Object`. Поэтому вы могли добавлять и `int`, и `string` *в одну и ту же коллекцию*, не получая при этом никаких сообщений об ошибках (в силу наследования и полиморфизма C#, о которых пойдет речь в части 2, "Объектно-ориентированное программирование на C#").

Но у такого решения имеются очень серьезные недостатки: чтобы получить, например, `int`, о котором известно, что он был помещен в коллекцию, требуется привести объект `Object` к типу `int`:

```
ArrayList ints = new ArrayList(); // Старый список Objects
int myInt = (int)ints[0];         // Получение первого int из списка
```

Это выглядит так, как будто `int` спрятано в матрешке. Если не выполнить приведения типа, матрешка не раскроется, и вы получите ошибки, связанные с тем, например, что `Object` не поддерживает операцию + или другие методы, свойства и операторы, которые вы ожидаете от `int`. Можно работать и с данными ограничениями, но такой код подвержен ошибкам и слишком многословен из-за переполнения приведениями (на считая определенных накладных расходов, связанных с так называемой упаковкой (boxing), которые могут существенно замедлить программу).

Если в коллекции содержатся разнородные объекты, например капуста и короли, ситуация только ухудшается. Теперь придется выполнить определенные действия для выяснения, что именно вы выловили из коллекции — капусту или короля, чтобы операция приведения была корректной.

При всех этих ограничениях новые обобщенные коллекции оказались глотком свежего воздуха. Они не требуют выполнения приведений, и вы всегда знаете, какой объект получили из коллекции, поскольку в одну коллекцию можно помещать объекты только одного типа. Но вы будете встречаться со старыми коллекциями в чужом коде, а иногда вам даже придется ими пользоваться (когда у вас будут очень веские причины для хранения в одной коллекции башмаков и сургуча).

ЗАПОМНИ!

Необобщенные коллекции можно найти в пространствах имен `System.Collections` и `System.Collections.Specialized`. Современные обобщенные коллекции находятся в пространстве имен `System.Collections.Generic`. (О пространствах имен и обобщенном программировании будет рассказываться в части 2, "Объектно-ориентированное программирование на C#".)

Глава 7
Работа с коллекциями

В ЭТОЙ ГЛАВЕ...

- Работа с каталогами и файлами как с коллекциями
- Перечисление элементов коллекций
- Реализация индексации для простого обращения к объектам коллекции
- Обход коллекции при помощи блока итератора C#

В главе 6, "Глава для коллекционеров", рассматриваются *классы коллекций* из библиотеки классов .NET Framework, используемые в C# и других языках программирования .NET.

В первой части данной главы расширяется понятие "коллекция". Например, рассмотрим следующие коллекции: файл как коллекцию строк или записей с данными и каталог как коллекцию файлов. Эта глава основана на материале главы 6, "Глава для коллекционеров", и материале о файлах из части 3, "Вопросы проектирования на C#".

Однако основное внимание здесь уделяется обходу, или *итерации*, коллекций всех видов — от каталогов файлов до всех видов массивов и списков.

Обход каталога файлов

В ряде случаев требуется в поисках чего-то просканировать каталог файлов. Приведенная ниже демонстрационная программа `LoopThroughFiles` просматривает все файлы в данном каталоге, считывая каждый файл и выводя его

содержимое на консоль в шестнадцатеричном формате. Это демонстрирует вам, что файл можно выводить не только в виде строк. (Что такое шестнадцатеричный формат, вы узнаете немного позже.)

Использование программы LoopThroughFiles

В командной строке пользователь указывает каталог, используемый в качестве аргумента программы. Следующая команда будет выводить в шестнадцатеричном виде все файлы из каталога `temp` (как бинарные, так и текстовые):

```
loopthroughfiles c:\temp
```

Если не ввести имя каталога, по умолчанию программа использует текущий каталог.

ВНИМАНИЕ!

Если запустить эту программу в каталоге с большим количеством файлов, то вывод шестнадцатеричного дампа может занять длительное время. Много времени требует и вывод дампа большого файла. Либо испытайте программу на каталоге покороче, либо, когда вам надоест, просто нажмите клавиши <Ctrl+C>. Эта команда должна прервать выполнение программы в любом консольном окне.

Если вы укажете некорректный каталог `x`, то вывод программы будет иметь следующий вид:

```
Каталог "x" неверен
Could not find a part of the path "C:\C#Programs\LoopThroughFiles\bin\
Debug\x".

Файлов больше нет
Нажмите <Enter> для завершения программы...
```

ШЕСТНАДЦАТЕРИЧНЫЕ ЧИСЛА

Как и бинарные числа (`0` и `1`), шестнадцатеричные числа также очень важны в компьютерном программировании. В шестнадцатеричной системе счисления цифрами являются обычные десятичные цифры `0-9` и буквы `A`, `B`, `C`, `D`, `E` и `F`, где `A` = 10, `B` = 11, ..., `F` = 15. Для иллюстрации (префикс `0x` указывает на шестнадцатеричность выводимого числа):

```
0xD = 13 decimal
0x10 = 16 decimal: 1*16 + 0*1
0x2A = 42 decimal: 2*16 + A*1 (здесь A*1 = 10*1)
```

Буквы могут быть как строчными, так и прописными: `F` означает то же, что и `f`. Эти числа выглядят причудливо, но они очень полезны, в особенности при отладке или работе с аппаратной частью или содержимым памяти.

Начало программы

Как и во всех примерах в книге, программа начинается с базовой структуры, показанной ниже. Обратите внимание, что вы должны включить отдельную директиву `using` для пространства имен `System.IO`. К этой базовой структуре необходимо добавить отдельные функции, описанные в следующих разделах.

```
using System;
using System.IO;
// LoopThroughFiles - проход по всем файлам, содержащимся в
// каталоге. Здесь выполняется вывод шестнадцатеричного
// дампа файла на экран, но могут выполняться и любые иные
// действия
namespace LoopThroughFiles
{
    public class Program
    {
    }
}
```

Получение начальных входных данных

Каждое консольное приложение начинается с функции `Main()`, как показано во всех предыдущих главах. Не волнуйтесь, если вы не совсем понимаете, как функция `Main()` должна работать в рамках консольного приложения. Пока просто знайте, что первой функцией, которую вызывает C#, является функция `Main()` консольного приложения, как показано в следующем коде.

```
public static void Main(string[] args) {
    // Если имя каталога не указано...
    string directoryName;

    if (args.Length == 0) {
        // ...получаем имя текущего каталога...
        directoryName = Directory.GetCurrentDirectory();
    } else {
        // ...в противном случае рассматриваем первый аргумент
        // в качестве имени рабочего каталога.
        directoryName = args[0];
    }

    Console.WriteLine(directoryName);
    // Получение списка файлов в каталоге.
    FileInfo[] files = GetFileList(directoryName);

    // Проход по всем файлам списка,
    // с выводом шестнадцатеричного дампа каждого файла.
    foreach (FileInfo file in files) {
        // Запись имени файла.
        Console.WriteLine("\n\nДамп файла {0}:",
                          file.FullName);
```

```
        // Вывод содержимого файла на консоль.
        DumpHex(file);
        // Ожидание перед выводом следующего файла.
        Console.WriteLine("\nНажмите <Enter> для продолжения");
        Console.ReadLine();
    }

    // Работа сделана!
    Console.WriteLine("Файлов больше нет");
    // Ожидание подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для "
                      "завершения программы...");
    Console.Read();
}
```

Первая строка демонстрационной программы `LoopThroughFiles` определяет наличие аргумента в командной строке. Если список аргументов пуст (`args.Length` равен 0), программа вызывает `Directory.GetCurrentDirectory()`. Если программа запущена из Visual Studio, а не из командной строки, то по умолчанию будет использоваться подкаталог `bin\Debug` в каталоге проекта `LoopThroughFiles`.

СОВЕТ

Класс `Directory` предоставляет пользователю набор методов для работы с каталогами, а класс `FileInfo` — для перемещения, копирования и удаления файлов.

Затем программа получает список всех файлов в указанном каталоге посредством вызова `GetFileList()`. Этот метод возвращает массив объектов `FileInfo`. Каждый объект `FileInfo` содержит информацию о файле, например имя файла (как полное имя с путем, `FullName`, так и без пути — `Name`), дату его создания и время последнего изменения. Метод `Main()` проходит по всему списку файлов с помощью цикла `foreach`. Он выводит имя каждого файла и передает его методу `DumpHex()` для вывода содержимого на консоль.

Создание списка файлов

Прежде чем начать работу со списком файлов, его необходимо создать. Метод `GetFileList()` начинает с создания пустого списка элементов `FileInfo`. Этот список возвращается в случае ошибки. Вот соответствующий код.

```
// GetFileList - получение списка всех файлов в
// указанном каталоге
public static FileInfo[]
GetFileList(string directoryName) {
    // Начинаем с пустого списка
    FileInfo[] files = new FileInfo[0];
```

```
    try {
        // Получаем информацию о каталоге
        DirectoryInfo di =
            new DirectoryInfo(directoryName);
        // В ней имеется список файлов
        files = di.GetFiles();
    } catch (Exception e) {
        Console.WriteLine("Каталог \"" + directoryName +
                          "\" неверен");
        Console.WriteLine(e.Message);
    }

    return files;
}
```

Затем метод `GetFileList()` создает объект `DirectoryInfo`. Как и гласит его имя, объект `DirectoryInfo` содержит тот же вид информации о каталоге, что и объект `FileInfo` о файле. Однако у объекта `DirectoryInfo` есть доступ к одной вещи, к которой нет доступа у объекта `FileInfo`, — к списку файлов каталога в виде массива `FileInfo`.

Как обычно, метод `GetFileList()` помещает код, работающий с файлами и каталогами, в большой `try`-блок. (Объяснение, что такое ключевые слова `try` и `catch`, дается в главе 9, "Эти исключительные исключения".) Конструкция `catch` в конце метода перехватывает все генерируемые ошибки и выводит имя каталога (которое, вероятно, введено неверно, т.е. такого каталога не существует).

ВНИМАНИЕ!

Последний шаг состоит в возврате `files`, который содержит список файлов. Будьте внимательны при возврате ссылок. Например, не возвращайте ссылки ни на одну из внутренних очередей в классе `PriorityQueue` в главе 8, "Обобщенность", если не хотите намеренно пригласить пользователей мешать нормальной работе класса (путем работы не через методы класса, а напрямую с очередями). Но `GetFileList()` не дает вам доступа к внутренностям одного из ваших классов, так что в данном случае все в порядке.

Форматирование вывода

С собранным списком файлов вы можете делать все что хотите. Приведенный пример отображает содержимое каждого файла в шестнадцатеричном формате, который может быть полезным в некоторых ситуациях. Перед тем как вы сможете создать строку вывода в шестнадцатеричном формате, вам нужно создать отдельные выходные строки. Метод `DumpHex()`, представленный здесь, может показаться вам сложным из-за трудностей правильного форматирования вывода.

```
// DumpHex для заданного файла выводит его содержимое
// на консоль
public static void DumpHex(FileInfo file)
{
  // Открываем файл
  FileStream fs;
  BinaryReader reader;
  try
  {
    fs = file.OpenRead();
    // Заворачиваем поток в BinaryReader.
    reader = new BinaryReader(fs);
  }
  catch(Exception e)
  {
    Console.WriteLine("\nНе могу читать \"" +
                      file.FullName + "\"");
    Console.WriteLine(e.Message);
    return;
  }

  // Построчный проход по содержимому файла
  for(int line = 1; true; line++)
  {
    // Считываем очередные 10 байт (это все, что можно
    // разместить в одной строке); выходим, когда все
    // байты считаны
    byte[] buffer = new byte[10];
    // Для чтения используем BinaryReader.
    // Примечание: в этом случае использование
    // FileStream также было бы очень простым делом.
    int numBytes = reader.Read(buffer, 0,
                               buffer.Length);
    if (numBytes == 0)
    {
      return;
    }
    // Выводим считанные только что данные, предваряя их
    // номером строки
    Console.Write("{0:D3} - ", line);
    DumpBuffer(buffer, numBytes);
    // После каждых 20 строк останавливаемся, так как
    // прокрутка консольного окна отсутствует
    if ((line % 20) == 0)
    {
      Console.WriteLine("Нажмите <Enter> для вывода " +
                        "очередных 20 строк");
      Console.ReadLine();
    }
  }
}
```

Метод `DumpHex()` начинает работу с открытия файла. Объект `FileInfo` содержит информацию о файле, но не открывает его. Метод `DumpHex()` получает полное имя файла, включая путь. Затем он открывает `FileStream` в режиме только для чтения с использованием этого имени. Блок `catch` перехватывает исключение, если `FileStream` не в состоянии прочесть файл по той или иной причине.

Затем `DumpHex()` считывает файл по 10 байт за раз и выводит их в одну строку в шестнадцатеричном формате. После вывода каждых 20 строк программа приостанавливает работу в ожидании нажатия пользователем клавиши <Enter>. При реализации этой функциональности я воспользовался оператором получения остатка от деления `%`.


СОВЕТ

По вертикали консольное окно по умолчанию имеет 25 строк (правда, пользователь может изменить эту настройку, добавив или убрав строки). Это означает, что вы должны делать паузу после вывода каждых 20 строк или около того. В противном случае данные будут быстро выведены на экран, и пользователь не сможет их прочесть.

Операция деления по модулю (`%`) возвращает остаток после деления, т.е. выражение `(line%20)==0` истинно при значениях `line`, равных 20, 40, 60, 80... Словом, идея понятна. Это важный метод, применимый для всех видов циклов, когда нужно выполнять некоторую операцию только с определенной частотой.

Вывод в шестнадцатеричном формате

Метод `DumpBuffer()` выводит каждый член массива байтов с использованием управляющего элемента форматирования `X2`. `X2`, хотя и звучит как название какого-то секретного военного эксперимента, означает всего лишь "вывести число в виде двух шестнадцатеричных цифр".

```
// DumpBuffer - вывод буфера символов в виде единой
// строки в шестнадцатеричном формате
public static void DumpBuffer(byte[] buffer,
                              int numBytes)
{
  for(int index = 0; index < numBytes; index++)
  {
    byte b = buffer[index];
    Console.Write("{0:X2}, ", b);
  }
  Console.WriteLine();
}
```

Диапазон значений `byte` — от 0 до 255, или 0xFF, т.е. двух шестнадцатеричных цифр для вывода одного байта достаточно. Вот как выглядят первые 20 строк при выводе содержимого файла `output.txt`.

```
Дамп файла C:\Temp\output.txt:
001 - 53, 74, 72, 65, 61, 6D, 20, 28, 70, 72,
002 - 6F, 74, 65, 63, 74, 65, 64, 29, 0D, 0A,
003 - 20, 20, 46, 69, 6C, 65, 53, 74, 72, 65,
004 - 61, 6D, 28, 73, 74, 72, 69, 6E, 67, 2C,
005 - 20, 46, 69, 6C, 65, 4D, 6F, 64, 65, 2C,
006 - 20, 46, 69, 6C, 65, 41, 63, 63, 65, 73,
007 - 73, 29, 0D, 0A, 20, 20, 4D, 65, 6D, 6F,
008 - 72, 79, 53, 74, 72, 65, 61, 6D, 28, 29,
009 - 3B, 0D, 0A, 20, 20, 4E, 65, 74, 77, 6F,
010 - 72, 6B, 53, 74, 72, 65, 61, 6D, 0D, 0A,
011 - 20, 20, 42, 75, 66, 66, 65, 72, 53, 74,
012 - 72, 65, 61, 6D, 20, 2D, 20, 62, 75, 66,
013 - 66, 65, 72, 73, 20, 61, 6E, 20, 65, 78,
014 - 69, 73, 74, 69, 6E, 67, 20, 73, 74, 72,
015 - 65, 61, 6D, 20, 6F, 62, 6A, 65, 63, 74,
016 - 0D, 0A, 0D, 0A, 42, 69, 6E, 61, 72, 79,
017 - 52, 65, 61, 64, 65, 72, 20, 2D, 20, 72,
018 - 65, 61, 64, 20, 69, 6E, 20, 76, 61, 72,
019 - 69, 6F, 75, 73, 20, 74, 79, 70, 65, 73,
020 - 20, 28, 43, 68, 61, 72, 2C, 20, 49, 6E,
Нажмите <Enter> для вывода очередных 20 строк
```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Можно восстановить файл в виде строк из вывода в шестнадцатеричном формате. `0x61` — числовой эквивалент символа *a*. Буквы расположены в алфавитном порядке, так что `0x65` должно быть символом *e*. `0x20` — пробел. Приведенная здесь первая строка выглядит при обычной записи в виде строк как "`Stream (pr`". Интригующе, не правда ли? Полностью коды букв вы можете найти, выполнив в Google поиск *ASCII table*.

Эти коды корректны и при использовании набора символов Unicode, который применяется C# по умолчанию (больше о Unicode вы можете узнать, про*гугл*явшись в поисках термина *Unicode characters*).

Обход коллекций: итераторы

В оставшейся части главы будут проанализированы три разных подхода к общей задаче *итерирования* коллекции. В этом разделе будет продолжено обсуждение наиболее традиционного (как минимум для программистов на C#) подхода с использованием итераторов, которые реализуют интерфейс `IEnumerator`.

СОВЕТ

Термины *итератор* (iterator) и *перечислитель* (enumerator) являются синонимами. Термин *итератор* более распространен, несмотря на имя реализуемого им интерфейса. От обоих терминов можно произвести глагольную форму — вы можете итерировать контейнер, а можете перечислять. Другими подходами к решению этой же задачи являются индексаторы и новые блоки итераторов.

Доступ к коллекции: общая задача

Различные типы коллекций могут иметь разные схемы доступа. Не все виды коллекций могут быть эффективно доступны с использованием индексов наподобие массивов, таков, например, связанный список. Различия между типами коллекций делают невозможным написание без специальных средств метода наподобие приведенного далее:

```
// Передается коллекция любого вида
void myClearMethod(Collection aColl, int index)
{
  aColl[index] = 0; // Индексирование работает не для всех
                    // типов коллекций
  // ...продолжение...
}
```

Коллекции каждого типа могут сами определять свои методы доступа (и делают это). Например, связанный список может предоставить метод `GetNext()` для выборки следующего элемента из цепочки объектов; стек может предложить методы `Push()` и `Pop()` для добавления и удаления объектов и т.д.

Более общий подход состоит в предоставлении для каждого класса коллекции отдельного так называемого *класса итератора*, который знает, как работать с конкретной коллекцией. Каждая коллекция `X` определяет собственный класс `IteratorX`. В отличие от `X`, `IteratorX` представляет общий интерфейс `IEnumerator`, золотой стандарт итерирования. Этот метод использует второй объект, именуемый *итератором*, в качестве указателя внутрь коллекции. Итератор (перечислитель) обладает следующими преимуществами.

- Каждый класс коллекции может определить собственный класс итератора. Поскольку итератор реализует стандартный интерфейс `IEnumerator`, с ним обычно легко работать.
- Прикладной код не должен знать о внутреннем устройстве коллекций. Пока программист работает с итератором, тот берет на себя все заботы о деталях. Это — хорошая инкапсуляция.
- Прикладной код может создать много независимых объектов-итераторов для одной и той же коллекции. Поскольку итератор содержит информацию о собственном состоянии (знает, где он находится

в процессе итерирования), каждый итератор может независимо проходить по коллекции. Вы можете одновременно выполнять несколько итераций, причем в один и тот же момент все они могут находиться в разных позициях.

Чтобы сделать возможной работу цикла `foreach`, интерфейс `IEnumerator` должен поддерживать различные типы коллекций — от массивов до связанных списков. Следовательно, его методы должны быть максимально обобщенными, насколько это возможно. Например, нельзя использовать итератор для произвольного доступа к элементам коллекции, поскольку большинство коллекций не обеспечивают подобного доступа. `IEnumerator` предоставляет три следующих метода.

- `Reset()` — устанавливает итератор таким образом, чтобы он указывал на начало коллекции. ***Примечание***: обобщенная версия `IEnumerator`, `IEnumerator<T>`, не предоставляет метод `Reset()`. В случае обобщенного `LinkedList` из .NET (находится в `System.Collections.Generic`) просто начинайте работу с вызова `MoveNext()`.
- `MoveNext()` — перемещает итератор от текущего объекта в контейнере к следующему.
- `Current` — свойство (не метод), которое дает объект данных, хранящийся в текущей позиции итератора.

Описанный принцип продемонстрирован приведенным ниже методом. Программист класса `MyCollection` (не показанного здесь) создает соответствующий класс итератора — скажем, `IteratorMyCollection` (применяя соглашение об именах `IteratorX`, упоминавшееся ранее). Прикладной программист ранее сохранил ряд объектов `ContainedDataObjects` в коллекции `MyCollection`. Приведенный ниже фрагмент исходного текста использует три стандартных метода `IEnumerator` для чтения этих объектов:

```
// Класс MyCollection хранит в качестве данных объекты типа
// ContainedDataObject
void MyMethod(MyCollection myColl)
{
  // Программист, создавший класс MyCollection, создал также
  // и класс итератора IteratorMyCollection; прикладной
  // программист создает объект итератора для прохода по
  // объекту myColl
  IEnumerator iterator = new IteratorMyCollection(myColl);
  // перемещаем итератор в "следующую позицию" внутри
  // коллекции
  while(iterator.MoveNext())
  {
```

```
    // Получаем ссылку на объект данных в текущей позиции
    // коллекции
    ContainedDataObject containedData; // data
    contained = (ContainedDataObject)iterator.Current;
    // ...используем объект данных contained...
  }
}
```

Метод `MyMethod()` принимает в качестве аргумента коллекцию `ContainedDataObject`. Он начинается с создания итератора типа `IteratorMyCollection`. Метод начинает цикл с вызова `MoveNext()`. При первом вызове `MoveNext()` перемещает итератор к первому элементу коллекции. При каждом последующем вызове `MoveNext()` перемещает указатель "на одну позицию". Метод `MoveNext()` возвращает `false`, когда коллекция исчерпана и итератор больше нельзя передвинуть.

Свойство `Current` возвращает ссылку на объект данных в текущей позиции итератора. Программа преобразует возвращаемый объект в `ContainedDataObject` перед тем, как присвоить его переменной `contained`. Вызов `Current` некорректен, если предшествующий вызов метода `MoveNext()` не вернул `true`.

Использование `foreach`

Методы `IEnumerator` достаточно стандартны для того, чтобы C# использовал их автоматически для реализации конструкции `foreach`. Цикл `foreach` может обращаться к любому классу, реализующему интерфейс `IEnumerable` или `IEnumerable<T>`, как показано в приведенном обобщенном методе, который может работать с любым классом — от массивов и связанных списков до стеков и очередей:

```
void MyMethod(IEnumerable<T> containerOfThings)
{
  foreach(string s in containerOfThings)
  {
    Console.WriteLine("Следующая строка - {0}", s);
  }
}
```

Класс реализует `IEnumerable<T>` путем определения метода `GetEnumerator()`, который возвращает экземпляр `IEnumerator<T>`. Скрыто от посторонних глаз `foreach` вызывает метод `GetEnumerator()` для получения итератора. Цикл использует этот итератор для обхода контейнера. Каждый выбираемый им элемент приводится к соответствующему типу перед тем, как продолжить выполнение тела цикла. Обратите внимание на то, что `IEnumer`**`able`** и `IEnumer`**`ator`** различные, но связанные интерфейсы. C#

предоставляет и необобщенную версию обоих интерфейсов, но для повышения безопасности типов следует предпочесть обобщенную версию.

`IEnumerable<T>` имеет следующий вид:

```
interface IEnumerable<T>
{
  IEnumerator<T> GetEnumerator();
}
```

А `IEnumerator<T>` — такой:

```
interface IEnumerator<T>
{
  bool MoveNext();
  T Current { get; }
}
```

Необобщенный интерфейс `IEnumerator` добавляет метод `Reset()`, который перемещает итератор в начало коллекции, а его свойство `Current` возвращает тип `Object`. Обратите внимание на то, что `IEnumerator<T>` унаследован от `IEnumerator`, и вспомните, что наследование интерфейсов (рассматривается в главе 18, "Интерфейсы") отличается от обычного наследования объектов.

Массивы C# (воплощенные в классе `Array`) и все классы коллекций .NET реализуют оба интерфейса. Поэтому беспокоиться о реализации этих интерфейсов следует только при разработке собственных коллекций. Для встроенных коллекций вы можете просто их использовать (см. раздел *System.Collections.Generic namespace* справочной системы). Итак, цикл `foreach` можно записать таким образом:

```
foreach(int nValue in myContainer)
{
  // ...
}
```

Обращение к коллекциям как к массивам: индексаторы

Обращение к элементам массива очень простое и понятное: команда `container[n]` обеспечивает обращение к *n*-му элементу массива `container`. Было бы хорошо, если бы так же просто с помощью индексов можно было обращаться и к другим типам коллекций.

Язык C# позволяет написать собственную реализацию операции индексирования. Вы можете предоставить возможность обращения через индекс коллекциям, которые таким свойством изначально не обладают. Кроме того, вы можете индексировать с использованием в качестве индексов не только типа `int`, но

и других типов, например string (как вам понравится возможность обращения container["Joe"]?). (В главе 22, "Структуры", показано, как добавить индексатор к struct.)

Формат индексатора

Индексатор выглядит очень похоже на обычное свойство, с тем исключением, что в нем вместо имени свойства появляются ключевое слово this и оператор индекса []:

```
class MyArray
{
  public string this[int index] // Обратите внимание на
  {                             // ключевое слово "this"
    get
    {
      return array[index];
    }
    set
    {
      array[index] = value;
    }
  }
}
```

За сценой выражение s = myArray[i]; вызывает метод доступа get, передавая ему значение индекса i. Выражение myArray[i] = "строка"; приводит к вызову метода доступа set, которому передаются индекс i и строка "строка" в качестве value.

Пример программы с использованием индексатора

Индексы не ограничены типом int. Например, вы можете использовать для индексации коллекции домов имена их владельцев или адреса. Кроме того, свойство индексатора могут быть перегружено с несколькими типами индексов, так что можно индексировать различные элементы одной и той же коллекции. Приведенная ниже демонстрационная программа Indexer генерирует класс виртуального массива KeyedArray, который выглядит и функционирует точно так же, как обычный массив, с тем исключением, что в качестве индекса применяется значение типа string.

Выполнение необходимой настройки класса

Этот пример основан на специальном классе, т.е. для него необходимо создать каркас класса. Вот каркас, используемый для хранения методов класса, которые обсуждаются в последующих разделах.

```
using System;

// Indexer - данная демонстрационная программа иллюстрирует
// применение оператора индекса для обеспечения доступа к
// массиву с использованием строк в качестве индексов
namespace Indexer
{
  public class KeyedArray
  {
    // Следующая строка обеспечивает "ключ" к массиву -
    // это строка, которая идентифицирует элемент
    private string[] _keys;

    // object представляет собой фактические данные,
    // связанные с ключом
    private object[] _arrayElements;

    // KeyedArray - создание KeyedArray фиксированного
    // размера
    public KeyedArray(int nSize)
    {
      _keys = new string[nSize];
      _arrayElements = new object[nSize];
    }
}
```

Класс `KeyedArray` включает два обычных массива. Массив `_arrayElements` содержит реальные данные `KeyedArray`. Строки, которые хранятся в массиве `_keys`, работают в качестве идентификаторов массива объектов, *i*-й элемент `_keys` соответствует *i*-й записи `_arrayElements`. Это позволяет прикладной программе индексировать `KeyedArray` с помощью индексов типа `string`. (Индексы, не являющиеся целыми числами, называются *ключами*.)

Строка `public KeyedArray(int size)` является началом особого рода функции, именуемой *конструктором*. Думайте о конструкторах как об инструкциях для создания экземпляров класса. Сейчас вам не нужно об этом беспокоиться, но на самом деле конструктор присваивает значения для `_keys` и `_arrayElements`.

Работа с индексаторами

Теперь нужно определить индексатор, который сделает код рабочим. Как это сделать, показано в следующем фрагменте исходного текста. Обратите внимание, что индексатор, `public object this[string key]`, требует использования двух функций, `Find()` и `FindEmpty()`.

```
// Find - поиск индекса записи, соответствующей строке
// targetKey (если запись не найдена, возвращает -1)
private int Find(string targetKey)
{
```

```
  for(int i = 0; i < _keys.Length; i++)
  {
    if (String.Compare(_keys[i], targetKey) == 0)
    {
      return i;
    }
  }
  return -1;
}

// FindEmpty - поиск свободного места в массиве для
// новой записи
private int FindEmpty()
{
  for (int i = 0; i < _keys.Length; i++)
  {
    if (_keys[i] == null)
    {
      return i;
    }
  }
  throw new Exception("Массив заполнен");
}

// Ищем содержимое по указанной строке - это и есть
// индексатор
public object this[string key]
{
  set
  {
    // Проверяем, нет ли уже такой строки
    int index = Find(key);
    if (index < 0)
    {
      // Если нет, ищем новое место
      index = FindEmpty();
      _keys[index] = key;
    }

    // Сохраняем объект в соответствующей позиции
    _arrayElements[index] = value;
  }

  get
  {
    int index = Find(key);
    if (index < 0)
    {
      return null;
    }
    return _arrayElements[index];
  }
}
```

Индексатор `set[string]` начинает с проверки, нет ли данного индекса в массиве, применяя метод `Find()`. Если он возвращает индекс, `set[]` сохраняет новый объект данных в соответствующем элементе `_arrayElements`. Если `Find()` не может найти ключ, `set[]` вызывает `FindEmpty()` для возврата пустого элемента, где и будет сохранен переданный объект.

Метод `get[]` работает с индексом с применением аналогичной логики. Сначала он ищет определенный ключ с использованием метода `Find()`. Если `Find()` возвращает неотрицательный индекс, `get[]` возвращает соответствующий член `_arrayElements`, в котором хранятся запрошенные данные. Если же `Find()` возвращает –1, то метод `get[]` возвращает значение `null`, указывающее, что переданный ключ в списке отсутствует.

Метод `Find()` циклически проходит по всем элементам массива `_keys` в поисках элемента с тем же значением, что и переданное значение `targetKey` типа `string`. Метод `Find()` возвращает индекс найденного элемента (или значение –1, если элемент не найден). Метод `FindEmpty()` возвращает индекс первого элемента, который не имеет связанного ключевого элемента.

Тестирование нового класса

Метод `Main()`, являющийся частью программы `Indexer`, но не частью класса, демонстрирует тривиальное применение класса `KeyedArray`:

```
public class Program
{
  public static void Main(string[] args)
  {
    // Создаем массив с достаточным количеством элементов
    KeyedArray ma = new KeyedArray(100);

    // Сохраняем возраст членов семьи Симпсонов
    ma["Bart"] = 8;
    ma["Lisa"] = 10;
    ma["Maggie"] = 2;

    // Ищем возраст Lisa
    Console.WriteLine("Ищем возраст Lisa");
    int age = (int)ma["Lisa"];
    Console.WriteLine("Возраст Lisa - {0}", age);

    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
  }
}
```

Сначала программа создает объект `ma` типа `KeyedArray` длиной 100 (т.е. со ста свободными элементами). Далее в этом объекте сохраняется возраст детей семьи Симпсонов с использованием имен в качестве индексов. И наконец программа получает возраст Лизы с применением выражения `ma["Lisa"]` и выводит его на экран.

Обратите внимание на то, что программа должна выполнить преобразование типа для значения, возвращенного из `ma[]`, так как `KeyedArray` написан таким образом, что может хранить объекты любого типа. Без такого преобразования типов можно обойтись, если индексатор написан так, что может работать только со значениями типа `int`, или если `KeyedArray` — обобщенный класс (см. главу 8, "Обобщенность"). Вывод программы прост и элегантен:

```
Ищем возраст Lisa
Возраст Lisa - 10
Нажмите <Enter> для завершения программы...
```

Блок итератора

В предыдущих версиях C# связанный список, обсуждавшийся в разделе "Обращение к коллекциям как к массивам: индексаторы" этой главы, был основным способом обхода коллекции, так же как в C++ и C. Хотя это решение вполне работоспособно, оказывается, что начиная с C# версии 2.0 этот процесс упрощен таким образом, что вам не нужно вызывать:

- `GetEnumerator()` (и выполнять преобразование типа результатов);
- `MoveNext()`;
- `Current` (и выполнять преобразование типа возвращаемого значения);
- Вы можете просто использовать `foreach` для обхода коллекции (C# сделает все остальное вместо вас).

  Честно говоря, `foreach` работает и для класса `LinkedList` из .NET. Это связано с наличием метода `GetEnumerator()`. Но я все еще должен самостоятельно писать класс `LinkedListIterator`. Новизна состоит в том, что вы можете пропустить при обходе часть своего класса.

Вместо реализации всех этих методов интерфейсов в создаваемых вами классах коллекций можно использовать *блоки итераторов* (iterator block) и обойтись без написания отдельного класса итератора для поддержки коллекций. Можно использовать блок итератора и для других рутинных работ.

Создание каркаса блока итератора

Наилучший способ реализовать итерирование — использовать блоки итераторов. Когда вы пишете класс коллекции, такой как `KeyedList` или `PriorityQueue`, вместо интерфейса `IEnumerator` вы реализуете блок итератора. Затем пользователи этого класса могут просто итерировать коллекцию с помощью цикла `foreach`. Вот как выглядит базовый каркас, содержащий функции, используемые в следующих разделах.

```
using System;
// IteratorBlocks - демонстрация применения блоков
// итераторов для написания итераторов коллекций
namespace IteratorBlocks
{
  class IteratorBlocks
  {
    //Main - демонстрация пяти различных приложений блоков
    // итераторов
    static void Main(string[] args)
    {
      // Итерирование месяцев года, вывод количества дней в
      // каждом из них
      MonthDays md = new MonthDays();
      // Итерируем
      Console.WriteLine("Месяцы:\n");
      foreach (string month in md)
      {
        Console.WriteLine(month);
      }
      // Инстанцируем коллекцию строк
      StringChunks sc = new StringChunks();
      // Итерируем - выводим текст, помещая каждый
      // фрагмент в собственной строке
      Console.WriteLine("\Строки:\n");
      foreach (string chunk in sc)
      {
        Console.WriteLine(chunk);
      }
      // А теперь выводим их в одну строку
      Console.WriteLine("\nВывод в одну строку:\n");
      foreach (string chunk in sc)
      {
        Console.Write(chunk);
      }
      Console.WriteLine();
      // Итерируем простые числа до 13
      YieldBreakEx yb = new YieldBreakEx();
      // Итерируем, останавливаясь после 13
      Console.WriteLine("\nПростые числа:\n");
      foreach (int prime in yb)
      {
        Console.WriteLine(prime);
      }
```

```
        // Итерируем четные числа в убывающем порядке
        EvenNumbers en = new EvenNumbers();
        // Вывод четных чисел от 10 до 4
        Console.WriteLine("\nЧетные числа:\n");
        foreach (int even in en.DescendingEvens(11, 3))
        {
          Console.WriteLine(even);
        }
        // Итерируем числа типа double
        PropertyIterator prop = new PropertyIterator();
        Console.WriteLine("\nЧисла double:\n");
        foreach (double db in prop.DoubleProp)
        {
          Console.WriteLine(db);
        }
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
      }
    }
```

Показанный здесь метод `Main()` обеспечивает основные функции тестирования блока кода итератора. Каждый из приведенных фрагментов кода демонстрирует, как код метода `Main()` взаимодействует с блоком итератора. Пока что достаточно просто знать, что метод `Main()` — это одна функция, и в следующих разделах мы разделим ее на части, чтобы ее было проще понять.

Итерирование дней в месяцах

Приведенный далее класс предоставляет итератор (показан полужирным шрифтом), который проходит по месяцам года.

```
// MonthDays - определяем итератор, который возвращает
// месяцы и количество дней в них
class MonthDays
{
  string[] months =
  { "January 31", "February 28", "March 31",
    "April 30", "May 31", "June 30", "July 31",
    "August 31", "September 30", "October 31",
    "November 30", "December 31" };
  //GetEnumerator - это и есть итератор
  public System.Collections.IEnumerator GetEnumerator()
  {
    foreach (string month in months)
    {
      // Возвращаем по одному месяцу в каждой итерации
      yield return month; // Новый синтаксис
    }
  }
}
```

Вот часть метода `Main()`, которая итерирует коллекцию с использованием цикла `foreach`.

```
// Итерирование месяцев года, вывод количества дней в
// каждом из них
MonthDays md = new MonthDays();
// Итерируем
Console.WriteLine("Месяцы:\n");
foreach (string month in md)
{
  Console.WriteLine(month);
}
```

Это простой класс коллекции, основанный на массиве, как и класс `KeyedArray`. Класс содержит массив, элементы которого имеют тип `string`. Когда клиент итерирует данную коллекцию, ее блок итератора выдает ему эти строки по одной. Каждая строка содержит имя месяца с количеством дней в нем. Здесь нет ничего сложного.

Класс определяет собственный блок итератора, в данном случае как метод `GetEnumerator()`. Метод `GetEnumerator()` возвращает объект типа `System.Collections.IEnumerator`. Да, вы должны были создавать такой метод и ранее, но вы должны были писать не только его, но и собственный класс-перечислитель для поддержки вашего класса-коллекции. Теперь же вы пишете только простой метод, возвращающий перечислитель с использованием новых ключевых слов `yield return`. Все остальное C# делает вместо вас: создает базовый класс-перечислитель и применяет его метод `MoveNext()` для итерирования. У вас уменьшаются количество работы и размер исходного текста.


ЗАПОМНИ!

Ваш класс, содержащий метод `GetEnumerator()`, больше не должен реализовывать интерфейс `IEnumerator`. В следующих разделах вам будет показано несколько вариаций блоков итераторов:

- обычные итераторы;
- именованные итераторы;
- свойства классов, реализованные как итераторы.

Обратите внимание на то, что метод `GetEnumerator()` класса `MonthDays` содержит цикл `foreach`, который работает со строками во внутреннем массиве. Блоки итераторов часто используют цикл того или иного вида, как вы увидите в примерах ниже в данной главе. Фактически в вашем коде имеется внутренний цикл `foreach`, который передает элемент за элементом в другой цикл `foreach` за пределами `GetEnumerator()`.

Что же такое коллекция

Остановимся на минутку и сравним эту небольшую коллекцию с коллекцией `LinkedList`, рассмотренной выше в главе. В то время как `LinkedList` имеет сложную структуру узлов, связанных посредством указателей, приведенная простейшая коллекция месяцев основана на простом массиве с фиксированным содержимым. Но понятие коллекции оказывается существенно шире.

Ваш класс коллекции не обязан иметь фиксированное содержимое — большинство коллекций разработаны для хранения объектов путем добавления их в коллекции, например, с помощью метода `Add()` или чего-то в этом роде. Класс `KeyedArray`, к примеру, использует для добавления элементов в коллекцию индексатор. Ваша коллекция также должна обеспечивать метод `Add()`, как и блок итератора, чтобы вы могли работать с ней с помощью цикла `foreach`.

Цель коллекции в наиболее общем смысле заключается в хранении множества объектов и обеспечении возможности их последовательного обхода по одному, хотя иногда может использоваться и произвольная выборка, как в демонстрационной программе `Indexer`. (Конечно, массив и так в состоянии справиться с этим, без дополнительных "наворотов" наподобие класса `MonthDays`, но итераторы вполне могут применяться и за пределами примера `MonthDays`.)

Говоря более обобщенно, независимо от того, что именно происходит за сценой, итерируемая коллекция генерирует "поток" значений, который можно получить с помощью `foreach`. Чтобы лучше понимать данную концепцию, ознакомьтесь с еще одним примером простого класса из демонстрационной программы `IteratorBlocks`, который иллюстрирует чистую идею коллекции:

```
//StringChunks - определение итератора, возвращающего
// фрагменты текста
class StringChunks
{
  //GetEnumerator - итератор. Обратите внимание
  // на то, как он (дважды) вызывается в Main
  public System.Collections.IEnumerator GetEnumerator()
  {
    // Возврат разных фрагментов текста на каждой итерации
    yield return "Using iterator ";
    yield return "blocks ";
    yield return "isn't all ";
    yield return "that hard";
    yield return ".";
  }
}
```

Коллекция `StringChunks`, как ни странно, ничего не *хранит* в обычном смысле этого слова. В ней нет даже массива. Так где же здесь коллекция? Она — в последовательности вызовов `yield return`, использующих специальный новый синтаксис для возврата элементов один за другим, пока все они не будут возвращены вызывающему методу. Эта коллекция "содержит" пять

объектов, каждый из которых представляет собой простую строку, как и в рассмотренном только что примере MonthDays. Извне класса, в методе Main(), вы можете итерировать эти объекты посредством простого цикла foreach, поскольку конструкция yield return возвращает по одной строке за раз. Вот часть метода Main(), в которой выполняется итерирование "коллекции" StringChunks:

```
// Инстанцируем коллекцию строк
StringChunks sc = new StringChunks();
// Итерируем - выводим текст, помещая каждый
// фрагмент в собственной строке
Console.WriteLine("\Строки:\n");
foreach (string chunk in sc)
{
  Console.WriteLine(chunk);
}
```

Синтаксис итератора

В C# 2.0 были введены два новых варианта синтаксиса итераторов. Конструкция yield return больше всего напоминает старую комбинацию MoveNext() и Current для получения очередного элемента коллекции. Конструкция yield break похожа на оператор break, который позволяет прекратить работу цикла или конструкции switch.

yield return

Синтаксис yield return работает следующим образом.

1. При первом вызове он возвращает первое значение коллекции.
2. При следующем вызове возвращается второе значение.
3. И так далее...

Это очень похоже на старый метод итератора MoveNext(), использовавшийся в коде LinkedList. Каждый вызов MoveNext() предоставляет новый элемент коллекции. Однако в данном случае вызов MoveNext() не требуется.

Что же подразумевается под "следующим вызовом"? Давайте еще раз посмотрим на цикл foreach, использующийся для итерирования коллекции StringChunks:

```
foreach (string chunk in sc)
{
  Console.WriteLine(chunk);
}
```

Каждый раз, когда цикл получает новый элемент посредством итератора, последний сохраняет достигнутую им позицию в коллекции. При очередной итерации цикла foreach итератор возвращает следующий элемент коллекции.

yield break

Следует упомянуть еще об одном синтаксисе. Можно остановить работу итератора в определенный момент, использовав в нем конструкцию `yield break`. Например, достигнут некоторый порог при тестировании определенного условия в блоке итератора класса коллекции и вы хотите на этом прекратить итерации. Вот краткий пример блока итератора, использующего `yield break` именно таким образом:

```
//YieldBreakEx - пример использования ключевого слова
// yield break
class YieldBreakEx
{
  int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
  //GetEnumerator - возврат последовательности простых
  // чисел с демонстрацией применения конструкции yield
  // break
  public System.Collections.IEnumerator GetEnumerator()
  {
    foreach (int prime in primes)
    {
      if (prime > 13) yield break; // Новый синтаксис
      yield return prime;
    }
  }
}
```

В рассмотренном случае блок итератора содержит оператор `if`, который проверяет все простые числа, возвращаемые итератором (кстати, с применением еще одного цикла `foreach` внутри итератора). Если простое число превышает 13, в блоке выполняется инструкция `yield break`, которая прекращает возврат простых чисел итератором. В противном случае работа итератора продолжалась бы и каждая инструкция `yield return` давала бы очередное простое число, пока коллекция полностью не исчерпалась бы.


СОВЕТ

Помимо использования в классах формальных коллекций для реализации перечислителей, любой из блоков итераторов в этой главе может быть написан как, например, статический метод в классе `Program`. Во многих случаях коллекция находится внутри метода. Такие коллекции специального назначения могут иметь много применений и обычно создаются очень быстро и просто.


СОВЕТ

Можно также написать *метод расширения* (рассматривается в части 2, "Объектно-ориентированное программирование на C#") для некоторого класса (или иного типа), который ведет себя, как блок итератора. Класс, который в определенном смысле может рассматриваться как коллекция, может оказаться очень полезным.

Блоки итераторов произвольного вида и размера

До этого момента блоки итераторов выглядели примерно следующим образом:

```
public System.Collections.IEnumerator GetEnumerator()
{
  yield return something;
}
```

Однако они могут принимать и другие формы:

- именованных итераторов и
- свойств классов.

Именованные итераторы

Вместо того чтобы писать блок итератора в виде метода с именем `GetEnumerator()`, можно написать *именованный итератор* — метод, возвращающий интерфейс `System.Collections.IEnumerable` вместо `IEnumerator`, который не обязан иметь имя `GetEnumerator()` (можете назвать его хоть `MyMethod()`). Вот, например, простой метод, который может использоваться для итерирования четных чисел от некоторого значения в порядке *убывания* до некоторого конечного значения, — да, да, именно в порядке убывания: для итераторов это сущие пустяки!

```
//EvenNumbers - определяет именованный итератор, который
// возвращает четные числа в определенном диапазоне в
// порядке убывания
class EvenNumbers
{
  //DescendingEvens - это "именованный итератор", в котором
  // используется ключевое слово yield break. Обратите
  // внимание на его использование в цикле foreach в
  // методе Main()
  public System.Collections.IEnumerable
        DescendingEvens(int top, int stop)
  {
    // Начинаем с ближайшего к top четного числа, не
    // превосходящего его
    if (top % 2 != 0)      // Если top нечетно
      top -= 1;
    // Итерации от top в порядке уменьшения до ближайшего к
    // stop четного числа, превосходящего его
    for (int i = top; i >= stop; i -= 2)
    {
      if (i < stop) yield break;
      // Возвращаем очередное четное число на каждой
      // итерации
      yield return i;
    }
  }
}
```

Метод `DescendingEvens()` получает два аргумента (удобная возможность), определяющих верхнюю и нижнюю границы выводимых четных чисел. Первое четное число равно первому аргументу или, если он нечетен, на 1 меньше него. Последнее генерируемое четное число равно значению второго аргумента `stop` (или, если `stop` нечетно, на 1 больше него). Этот метод возвращает не значение типа `int`, а интерфейс `IEnumerable`. Но в нем все равно имеется инструкция `yield return`, которая возвращает четное число и затем ожидает очередного вызова из цикла `foreach`.


ЗАПОМНИ!

Это еще один пример "коллекции", в основе которой нет никакой "настоящей" коллекции наподобие уже рассматривавшегося ранее класса `StringChunks`. Заметим также, что эта коллекция *вычисляется* — на этот раз возвращаемые значения не жестко закодированы, а вычисляются по мере необходимости. Это еще один способ получить коллекцию без коллекции. (Вы можете получать элементы коллекции откуда угодно, например из базы данных или от веб-сервиса.) И наконец, в этом примере демонстрируется, что вы можете итерировать так, как вам заблагорассудится, например с шагом –2, а не со стандартным единичным.


СОВЕТ

Итератор не обязан быть конечным. Рассмотрим следующий итератор, который при запросе выдает новое число:

```
public System.Collections.IEnumerable PositiveIntegers()
{
  for (int i = 0; ; i++)
  {
    yield return i;
  }
}
```


ВНИМАНИЕ!

Это, по сути, бесконечный цикл. Вы можете захотеть передать значение, которое должно остановить итерации. Вот пример того, как можно вызвать `DescendingEvens()` в цикле `foreach` в методе `Main()` (вызов приведенного выше `PositiveIntegers()` выполняется аналогично). Здесь заодно показано, что произойдет, если вы передадите в качестве граничных нечетные значения — еще одно применение оператора `%`:

```
// Инстанцирование класса "коллекции" EvenNumbers
EvenNumbers en = new EvenNumbers();
// Итерирование: выводим четные числа от 10 до 4
Console.WriteLine("\nПоток убывающих четных чисел:");
foreach(int even in en.DescendingEvens(11, 3))
{
  Console.WriteLine(even);
}
```

Этот вызов дает список четных чисел от 10 до 4. Обратите также внимание на то, как используется цикл `foreach`. Вы должны инстанцировать объект `EvenNumbers` (класс коллекции). Затем в инструкции `foreach` вызывается метод именованного итератора:

```
EvenNumbers en = new EvenNumbers();
foreach(int even in en.DescendingEvens(top, stop)) ...

EvenNumbers en = new EvenNumbers();
foreach(int even in en.DescendingEvens(nTop, nStop)) ...
```


СОВЕТ

Если бы `DescendingEvens()` был статическим методом, можно было обойтись без экземпляра класса. В этом случае его можно было бы вызвать с использованием имени класса, как обычно:

```
foreach(int even in EvenNumbers.DescendingEvens(nTop, nStop)) ...
```

Поток идей для потоков объектов

Теперь, когда вы можете сгенерировать "поток" четных чисел таким образом, подумайте о массе других полезных вещей, потоки которых вы можете получить с помощью аналогичных "коллекций" специального назначения: потоки степеней двойки, членов арифметических или геометрических прогрессий, простых чисел или чисел Фибоначчи — да что угодно. Как вам идея потока случайных чисел (чем, собственно, и занимается класс `Random`) или сгенерированных случайным образом объектов?

Итерируемые свойства

Можно также реализовать блок итератора в виде *свойства* класса, конкретнее — в методе доступа `get()` свойства. Вот простой класс со свойством `DoubleProp`. Метод доступа `get()` этого класса работает как блок итератора, возвращающий поток значений типа `double`:

```
// PropertyIterator - демонстрирует реализацию метода
// доступа get свойства класса как блока итератора
class PropertyIterator
{
  double[] doubles = { 1.0, 2.0, 3.5, 4.67 };
  // DoubleProp - свойство "get" с блоком итератора
  public System.Collections.IEnumerable DoubleProp
  {
    get
    {
      foreach(double db in doubles)
      {
        yield return db;
      }
    }
  }
}
```

Заголовок `DoubleProp` пишется так же, как и заголовок метода `DescendingEvens()` в примере именованного итератора. Он возвращает интерфейс `IEnumerable`, но в виде свойства, не использует скобок после имени свойства и имеет только метод доступа `get()`, но не `set()`. Метод доступа `get()` реализован как цикл `foreach`, который итерирует коллекцию и применяет стандартную инструкцию `yield return` для поочередного возврата элементов из коллекции чисел типа `double`. Вот как это свойство можно использовать в методе `Main()`:

```
// Инстанцируем класс "коллекции" PropertyIterator
PropertyIterator prop = new PropertyIterator();
// Итерируем ее: генерируем значения типа double по одному
foreach (double db in prop.DoubleProp)
{
  Console.WriteLine(db);
}
```

СОВЕТ

Вы можете использовать *обобщенные итераторы*. Подробнее они рассмотрены в справочной системе, в разделе, посвященном применению итераторов.

Глава 8

Обобщенность

В ЭТОЙ ГЛАВЕ...

- Обобщенный код — отличное решение
- Написание собственного обобщенного класса
- Написание обобщенных методов
- Использование обобщенных интерфейсов и делегатов

Проблема с коллекциями заключается в том, что вам нужно точно знать, что вы в них отправляете. Можете ли вы представить себе рецепт, который допускает только точно перечисленные ингредиенты, и никакие другие? Никаких замен — ничто даже не может быть названо по-другому! Именно так поступает большинство коллекций, но только не обобщенных.

Как и в случае с рецептами в аптеке, вы можете сэкономить, выбрав универсальную версию (дженерик)[1]. *Обобщенность* вошла в язык в версии C# 2.0 и представляет собой классы, методы, интерфейсы и делегаты, являющиеся "форматированными бланками", которые заполняются затем действительными типами. Например, класс `List<T>` определяет обобщенный список, очень похожий на старый необобщенный `ArrayList`, но гораздо лучше! Когда вы "вытаскиваете из раковины" `List<T>`, чтобы инстанцировать собственный список, например чисел типа `int`, то просто заменяете `T` на `int`:

```
List<int> myList = new List<int>(); // Список чисел int
```

[1] Игра слов: "generic" — "обобщенный класс"; еще один смысл слова "дженерик" — "лекарство-копия, которое совпадает с оригиналом по количеству действующего вещества и влиянию на организм, обычно гораздо дешевле патентованных аналогов". — *Примеч. пер.*

Самое ценное в таком списке то, что вы можете инстанцировать `List<T>` для *любого единого* типа данных (`string`, `Student`, `BankAccount`, `CorduroyPants`; словом, для любого) и все равно получить безопасный и надежный список без ошибок, свойственных необобщенным спискам.

Обобщенность в C# есть двух разновидностей: встроенные обобщенные классы типа `List<T>` и классы, разработанные программистами для решения собственных задач. После беглого обзора концепций обобщенности мы рассмотрим в данной главе, как создавать собственные обобщенные классы, методы, интерфейсы и делегаты.

Обобщенность в C#

Так зачем же нужна обобщенность? Основных причин две: безопасность и производительность.

Обобщенные классы безопасны


ЗАПОМНИ!

Объявляя массив, вы должны указать точный тип данных, которые могут в нем храниться. Если это `int`, то массив не может хранить ничего, кроме `int` или других числовых типов, которые C# в состоянии неявно преобразовать в `int`. Если вы попытаетесь поместить в массив данные неверного типа, то получите от компилятора сообщение об ошибке. Таким образом компилятор обеспечивает *безопасность типов*, т.е. вы обнаруживаете и исправляете проблему еще до того, как она проявится.

Гораздо лучше получить сообщение об ошибке от компилятора, чем в процессе работы программы, — ошибка компиляции позволяет проще разобраться, в чем у вас проблема.


ЗАПОМНИ!

Старые необобщенные коллекции небезопасны. В C# переменная любого типа ЯВЛЯЕТСЯ `Object`, поскольку класс `Object` является базовым классом для всех других типов, как типов-значений, так и типов-ссылок. Однако когда вы сохраняете *типы-значения* (числа, `char`, `bool`, `struct`) в коллекции, они должны быть *упакованы* при помещении в нее и *распакованы* при извлечении из нее. Ссылочные типы, такие как `string`, `Student` и `BankAccount`, упаковки-распаковки не требуют.

Первое следствие небезопасности необобщенных классов заключается в том, что вам требуется приведение типов (как показано в следующем

фрагменте исходного текста) для получения исходного объекта из ArrayList, так как этот тип скрыт внутри Object.

```
ArrayList aList = new ArrayList();
// Добавляем пять-шесть элементов, а затем...
string myString = (string)aList[4]; // преобразуем в string.
```

ВНИМАНИЕ!

Второе следствие в том, что в ArrayList *одновременно* могут храниться *объекты разных типов*. То есть вы можете написать, например, такой исходный текст:

```
ArrayList aList = new ArrayList();
aList.Add("a string");  // string  -- OK
aList.Add(3);           // int     -- OK
aList.Add(aStudent);    // Student -- OK
```

Однако если вы поместите в ArrayList (или другую необобщенную коллекцию) объекты разных несовместимых типов, то как вы потом сможете узнать тип, например, третьего элемента? Если это Student, а вы попытаетесь преобразовать его в string, то получите ошибку времени выполнения программы.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Для безопасности следует производить проверку с использованием оператора is (рассматривается в части 2, "Объектно-ориентированное программирование на C#") или альтернативного оператора as следующим образом:

```
// Проверяем, верный ли тип объекта, а затем приводим его...
if(aList[i] is Student)                    // Объект - Student?
{
    Student aStudent = (Student)aList[i]; // Да, преобразуем.
}

// Или выполняем преобразование и смотрим, что получилось...
Student aStudent = aList[i] as Student;    // Получаем Student.
if(aStudent != null)                       // Это невозможно,
{                                          // возврат null.
    // OK, можно работать с aStudent.
}
```

Избавиться от лишней работы можно с помощью обобщенных классов. Обобщенные коллекции работают, как и массивы: вы определяете один и только один тип, который может храниться в коллекции при ее объявлении.

Обобщенные классы эффективны

Полиморфизм позволяет типу Object хранить любой другой тип. Однако за такое удобство приходится платить упаковкой и распаковкой типов-значений (числа, char, bool, struct) при размещении их в необобщенных коллекциях

(более подробно о полиморфизме рассказывается в части 2, "Объектно-ориентированное программирование на C#").

Упаковка не так уж снижает эффективность, если ваша коллекция мала. Но если вы перемещаете тысячи или даже миллионы целых чисел типа `int` в необобщенной коллекции, это может отнять примерно в 20 раз больше времени (и потребовать дополнительной памяти) по сравнению с хранением объектов ссылочного типа. Упаковка также может привести к ошибкам, которые трудно обнаружить. Обобщенные коллекции незнакомы с проблемами, связанными с упаковкой и распаковкой.

Создание собственного обобщенного класса

Помимо встроенных обобщенных классов коллекций, C# позволяет писать собственные обобщенные классы — как коллекции, так и другие типы классов. Главное, что вы имеете возможность создать обобщенные версии классов, которые спроектированы *вами*.

Определение обобщенного класса переполнено записями `<T>`. Инстанцируя такой класс, вы указываете тип, который заменит `T` так же, как и в случае рассмотренных обобщенных коллекций. Посмотрите, насколько схожи приведенные ниже объявления:

```
// Встроенный класс:
LinkedList<int> aList = new LinkedList<int>();
// Пользовательский класс:
MyClass<int> aClass = new MyClass<int>();
```

Оба являются инстанцированиями классов: одно — встроенного, второе — пользовательского. Не каждый класс имеет смысл делать обобщенным, но далее в главе будет рассмотрен пример класса, который следует сделать именно таковым.

ЗАПОМНИ!

Классы, которые логически могут делать одни и те же вещи с данными разных типов, — наилучшие кандидаты в обобщенные классы. Наиболее типичным примером являются коллекции, способные хранить различные данные. Если в какой-то момент у вас появляется мысль "А ведь мне придется написать версию этого класса еще и для объектов `Student`", вероятно, ваш класс стоит сделать обобщенным.

Чтобы показать, как пишутся собственные обобщенные классы, будет разработан обобщенный класс для очереди специального вида, а именно — для *очереди с приоритетами*.

Очередь посылок

Представим себе почтовую контору наподобие FedEx. В дверь OOPs, Inc. поступает постоянный поток пакетов, которые надо доставить получателям. Однако пакеты не равны по важности: одни из них следует доставить немедленно (для них уже ведутся разработки телепортаторов), другие можно доставить грузовыми голубями, а третьи могут быть доставлены наземным транспортом.

Однако в контору пакеты приходят в произвольном порядке, так что при поступлении очередного пакета его нужно поставить в очередь на доставку. Слово прозвучало: необходима очередь, но очередь необычная. Вновь прибывшие пакеты становятся в очередь на доставку, но часть из них имеет более высокий приоритет и должна ставиться если и не в самое начало очереди, то уж точно не в ее конец. За исключением приоритетов, данная модель тютелька в тютельку укладывается в структуру данных *очереди*. Нам надо превратить ее в очередь с приоритетами.

Сценарий складской отгрузки аналогичен: новые посылки приходят и уходят в конец очереди — как правило. Но поскольку у некоторых есть более высокие приоритеты, они являются привилегированными посылками, как люди премиум-класса у стойки регистрации в аэропорту. Они могут проходить вперед, либо становясь первыми в очереди, либо располагаясь где-то недалеко от ее головы.

Очередь с приоритетами

Попробуем сформулировать правила очереди с приоритетами. Итак, в OOPs, Inc. входящие пакеты могут иметь высокий, средний и низкий приоритет. Ниже описан порядок их обработки.

- **Пакеты с высоким приоритетом** помещаются в начало очереди, но после других пакетов с высоким приоритетом, уже присутствующих в ней.
- **Пакеты со средним приоритетом** ставятся в начало очереди, но после пакетов с высоким приоритетом и других пакетов со средним приоритетом, уже присутствующих в ней.
- **Пакеты с низким приоритетом** ставятся в конец очереди.

Язык C# предоставляет встроенный обобщенный класс очереди, но он не подходит для создания очереди с приоритетами. Таким образом, нужно написать собственный класс очереди, но как это сделать? Распространенный подход заключается в разработке *класса-оболочки* (wrapper class) для нескольких очередей:

```
class Wrapper // Или PriorityQueue
{
    Queue _queueHigh   = new Queue ();
    Queue _queueMedium = new Queue ();
    Queue _queueLow    = new Queue ();
    // Методы для работы с этими очередями...
```

Оболочки (wrapper) — это классы (или методы), которые инкапсулируют в себе всю сложность. Оболочка может иметь интерфейс, существенно отличающийся от интерфейса того, что находится внутри нее — тогда это *адаптер* (adapter).

Оболочка инкапсулирует три обычные очереди (которые могут быть обобщенными) и управляет внесением пакетов в эти очереди и получением их из очередей. Стандартный интерфейс класса `Queue`, реализованного в C#, содержит два ключевых метода:

- `Enqueue()` — для помещения объектов *в конец* очереди;
- `Dequeue()` — для извлечения объектов *из начала* очереди.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В данном случае интерфейс оболочки совпадает с интерфейсом обычной очереди, так что ее можно рассматривать как обычную очередь. Класс реализует метод `Enqueue()`, который получает пакет и его приоритет, и на основании приоритета принимает решение о том, в какую из внутренних очередей его поместить. Он также реализует метод `Dequeue()`, который находит пакет с наивысшим приоритетом в своих внутренних очередях и извлекает его из очереди. Дадим рассматриваемому классу-оболочке формальное имя `PriorityQueue`. Вот полный исходный текст этого класса, чтобы вы могли посмотреть, как все это работает вместе. Ниже будет рассмотрена работа отдельных частей данного класса.

```
// PriorityQueue - демонстрация использования объектов
// низкоуровневой очереди для реализации высокоуровневой
// обобщенной очереди, в которой объекты хранятся с учетом
// их приоритета
using System;
using System.Collections.Generic;
namespace PriorityQueue
{
  class Program
  {
    //Main - заполняем очередь с приоритетами пакетами,
    // затем извлекаем из очереди их случайное количество
    static void Main(string[] args)
    {
      Console.WriteLine("Создание очереди с приоритетами:");
      PriorityQueue<Package> pq =
```

```
            new PriorityQueue<Package>();
        Console.WriteLine("Добавляем случайное количество" +
                          " (0 - 20) случайных пакетов" +
                          " в очередь:");
        Package pack;
        PackageFactory fact = new PackageFactory();
        // Нам нужно случайное число, которое меньше 20
        Random rand = new Random();
        // Случайное число в диапазоне 0-20
        int numToCreate = rand.Next(20);
        Console.WriteLine("\tСоздание {0} пакетов: ",
                          numToCreate);
        for (int i = 0; i < numToCreate; i++)
        {
          Console.Write("\t\tГенерация и добавление " +
                        "случайного пакета {0}", i);
          pack = fact.CreatePackage();
          Console.WriteLine(" с приоритетом {0}",
                            pack.Priority);
          pq.Enqueue(pack);
        }
        Console.WriteLine("Что получилось:");
        int total = pq.Count;
        Console.WriteLine("Получено пакетов: {0}", total);
        Console.WriteLine("Извлекаем случайное количество" +
                          " пакетов: 0-20: ");
        int numToRemove = rand.Next(20);
        Console.WriteLine("\tИзвлекаем {0} пакетов",
                          numToRemove);
        for (int i = 0; i < numToRemove; i++)
        {
          pack = pq.Dequeue();
          if (pack != null)
          {
            Console.WriteLine("\t\tДоставка пакета " +
                              "с приоритетом {0}",
                              pack.Priority);
          }
        }
        // Сколько пакетов "доставлено"
        Console.WriteLine("Доставлено {0} пакетов",
                          total - pq.Count);
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
      }
    }
    //Priority - вместо числовых приоритетов наподобие
    // 1, 2, 3 ... используем приоритеты с именами
    enum Priority // Об enum мы поговорим позже
    {
      Low, Medium, High
```

```
}
// IPrioritizable - определяем пользовательский интерфейс:
// классы, которые могут быть добавлены в PriorityQueue,
// должны реализовывать этот интерфейс
interface IPrioritizable
{
  // Пример свойства в интерфейсе
  Priority Priority { get; }
}
//PriorityQueue - обобщенный класс очереди с приоритетами;
// типы данных, добавляемых в очередь, обязаны
// реализовывать интерфейс IPrioritizable
class PriorityQueue<T>
        where T : IPrioritizable
{
  //Queues - три внутренние (обобщенные!) очереди
  private Queue<T> _queueHigh   = new Queue<T>();
  private Queue<T> _queueMedium = new Queue<T>();
  private Queue<T> _queueLow    = new Queue<T>();
  //Enqueue - добавляет T в очередь в соответствии с
  // приоритетом
  public void Enqueue(T item)
  {
    switch (item.Priority) // Требует реализации
    {                      // IPrioritizable
    case Priority.High:
      _queueHigh.Enqueue(item);
      break;
    case Priority.Low:
      _queueLow.Enqueue(item);
      break;
    case Priority.Medium:
      _queueMedium.Enqueue(item);
      break;
    default:
      throw new
        ArgumentOutOfRangeException(
             item.Priority.ToString(),
             "Неверный приоритет в PriorityQueue.Enqueue");
    }
  }
  //Dequeue - извлечение T из очереди с наивысшим
  // приоритетом
  public T Dequeue()
  {
    // Просматриваем очередь с наивысшим приоритетом
    Queue<T> queueTop = TopQueue();
    // Очередь не пуста
    if (queueTop != null && queueTop.Count > 0)
    {
      return queueTop.Dequeue(); // Возвращаем первый
                                 // элемент
    }
```

```
    // Если все очереди пусты, возвращаем null (здесь
    // можно сгенерировать исключение)
    return default(T); // Что это — мы рассмотрим позже
  }
  //TopQueue - непустая очередь с наивысшим приоритетом
  private Queue<T> TopQueue()
  {
    if (_queueHigh.Count > 0)    // Очередь с высоким
      return _queueHigh;         // приоритетом пуста?
    if (_queueMedium.Count > 0) // Очередь со средним
      return _queueMedium;       // приоритетом пуста?
    if (_queueLow.Count > 0)     // Очередь с низким
      return _queueLow;          // приоритетом пуста?
    return _queueLow;            // Все очереди пусты
  }
  //IsEmpty - Проверка, пуста ли очередь
  public bool IsEmpty()
  {
    // true, если все очереди пусты
    return (_queueHigh.Count   == 0) &
           (_queueMedium.Count == 0) &
           (_queueLow.Count    == 0);
  }
  //Count - Сколько всего элементов во всех очередях?
  public int Count // Реализуем как свойство только
  {                // для чтения
    get { return _queueHigh.Count   +
                 _queueMedium.Count +
                 _queueLow.Count; }
  }
}
// Package - пример класса, который может быть
// размещен в очереди с приоритетами
class Package : IPrioritizable
{
  private Priority priority;
  // Конструктор
  public Package(Priority priority)
  {
    this.priority = priority;
  }
  //Priority - возвращает приоритет пакета;
  // только для чтения
  public Priority Priority
  {
    get { return priority; }
  }
  // А также методы ToAddress, FromAddress, Insurance,
  // и другие...
}
// PackageFactory - класс, который знает, как создать
// новый пакет Package любого требуемого типа;
// такой класс называется классом-фабрикой.
```

```
class PackageFactory
{
  // Генератор случайных чисел.
  Random _randGen = new Random();

  // CreatePackage - метод фабрики, который выбирает
  // случайный приоритет и затем создает пакет с этим
  // приоритетом. Может быть реализован как блок
  // итератора.
  public Package CreatePackage()
  {
    // Случайным образом выбранный приоритет пакета.
    //  Может иметь значение 0, 1 или 2 (значения,
    // которые меньше 3).
    int rand = _randGen.Next(3);
    // Используем для генерации нового пакета; приведение
    // типа позволяет использовать значение в конструкции
    // switch.
    return new Package((Priority)rand);
  }
 }
}
```

СОВЕТ

Запустите программу `PriorityQueue` несколько раз. Поскольку в ней используется генератор случайных чисел, всякий раз она будет давать немного различающиеся результаты.

Распаковка пакета

Класс `Package` преднамеренно очень прост и написан исключительно для данной демонстрационной программы. Основное в нем — часть с приоритетом, хотя реальный класс `Package`, несомненно, должен содержать массу других членов. Вот соответствующий код:

```
// Package - пример класса, который может быть размещен в
// очереди с приоритетами. Любой класс, который реализует
// IPrioritizable, будет выглядеть похожим на Package.
class Package : IPrioritizable
{
    private Priority priority;
    // Конструктор
    public Package(Priority priority)
    {
        this.priority = priority;
    }
    //Priority - возвращает приоритет пакета; только для
    // чтения
    public Priority Priority
    {
        get { return priority; }
    }
```

```
    // А также методы ToAddress, FromAddress, Insurance,
    // и другие...
}
```

Все, что требуется классу `Package` для участия в данном пакете, — это

- член-данные для хранения приоритета,
- конструктор для создания пакета с определенным приоритетом,
- метод (реализованный здесь как свойство только для чтения) для возврата значения приоритета.

Требуют пояснения два аспекта класса `Package`: тип приоритета и интерфейс `IPrioritizable`, реализуемый данным классом.

Определение возможных приоритетов

Приоритеты представляют собой перечислимый тип (`enum`) под названием `Priority`. Он выглядит следующим образом:

```
//Priority - вместо числовых приоритетов наподобие
// 1, 2, 3 ... используем приоритеты с именами
enum Priority
{
    Low, Medium, High
}
```

Реализация интерфейса `IPrioritizable`

Любой объект, поступающий в `PriorityQueue`, должен знать собственный приоритет (общий принцип объектно-ориентированного программирования гласит, что каждый объект отвечает сам за себя).

СОВЕТ

Можно просто неформально "пообещать", что класс `Package` будет иметь член для получения его приоритета, но лучше заставить компилятор проверять это требование, т.е. то, что у любого объекта, помещаемого в `PriorityQueue`, есть этот член. Один из способов обеспечить это состоит в требовании, чтобы все объекты реализовывали интерфейс `IPrioritizable`:

```
// IPrioritizable - определяем пользовательский интерфейс:
// классы, которые могут быть добавлены в PriorityQueue,
// должны реализовывать этот интерфейс
interface IPrioritizable
{
    Priority Priority { get; }
}
```

Запись `{get;}` определяет, как должно быть описано свойство в объявлении интерфейса (см. главу 18, "Интерфейсы"). Класс `Package` реализует интерфейс путем предоставления реализации свойства `Priority`:

```
public Priority Priority
{
    get { return _priority; }
}
```

Вы встретитесь с другой стороной этого обязательного требования в объявлении класса `PriorityQueue` в разделе "И наконец — обобщенная очередь с приоритетами".

Метод `Main()`

Перед тем как приступить к исследованию класса `PriorityQueue`, стоит посмотреть, как он применяется на практике. Вот исходный текст метода `Main()`:

```
// Main - заполняем очередь с приоритетами пакетами,
// затем извлекаем из очереди их случайное количество
static void Main(string[] args)
{
  Console.WriteLine("Создание очереди с приоритетами:");
  PriorityQueue<Package> pq =
          new PriorityQueue<Package>();
  Console.WriteLine("Добавляем случайное количество" +
                    " (0-20) случайных пакетов" +
                    " в очередь:");
  Package pack;
  PackageFactory fact = new PackageFactory();
  // Нам нужно случайное число, которое меньше 20
  Random rand = new Random();
  // Случайное число в диапазоне 0-20
  int numToCreate = rand.Next(20);
  Console.WriteLine("\tСоздание {0} пакетов: ",
                    numToCreate);
  for (int i = 0; i < numToCreate; i++)
  {
    Console.Write("\t\tГенерация и добавление " +
                  "случайного пакета {0}", i);
    pack = fact.CreatePackage();
    Console.WriteLine(" с приоритетом {0}",
                      pack.Priority);
    pq.Enqueue(pack);
  }
  Console.WriteLine("Что получилось:");
  int total = pq.Count;
  Console.WriteLine("Получено пакетов: {0}", total);
  Console.WriteLine("Извлекаем случайное количество" +
                    " пакетов: 0-20: ");
  int numToRemove = rand.Next(20);
  Console.WriteLine("\tИзвлекаем {0} пакетов",
                    numToRemove);
  for (int i = 0; i < numToRemove; i++)
  {
    pack = pq.Dequeue();
```

```
    if (pack != null)
    {
      Console.WriteLine("\t\tДоставка пакета " +
                        "с приоритетом {0}",
                        pack.Priority);
    }
  }
  // Сколько пакетов "доставлено"
  Console.WriteLine("Доставлено {0} пакетов",
                    total - pq.Count);
  // Ожидаем подтверждения пользователя
  Console.WriteLine("Нажмите <Enter> для " +
                    "завершения программы...");
  Console.Read();
}
```

Итак, что же происходит в методе `Main()`?

1. Инстанцируется объект `PriorityQueue` для типа `Package`.
2. Создается объект `PackageFactory`, работа которого состоит в формировании новых пакетов со случайно выбранными приоритетами.

   Фабрика — это класс или метод, который создает для вас объекты. Ниже в этой главе вы поближе познакомитесь с фабрикой `PackageFactory`.
3. Для генерации случайного числа используется класс `Random` из библиотеки .NET, а затем вызывается `PackageFactory` для создания соответствующего количества новых объектов `Package` со случайными приоритетами.
4. Выполняется добавление созданных пакетов в `PriorityQueue` с помощью вызова `pq.Enqueue(pack)`.
5. Выводится число созданных пакетов, после чего некоторое случайное их количество извлекается из `PriorityQueue` с помощью вызова `pq.Dequeue()`.
6. Метод завершается выводом количества извлеченных из `PriorityQueue` пакетов.

Написание обобщенного кода

Как же написать собственный обобщенный класс со всеми этими `<T>`? Выглядит это, конечно, устрашающе, но все не так уж и страшно, что и демонстрирует данный раздел.

СОВЕТ

Простейший путь написания обобщенного класса состоит в создании сначала его необобщенной версии, а затем расстановки в ней всех этих `<T>`. Так, например, вы можете написать класс `PriorityQueue` для объектов `Package`, протестировать его, а затем "обобщить". Вот небольшая часть необобщенного класса `PriorityQueue` для иллюстрации сказанного:

```
public class PriorityQueue
{
    //Queues - три внутренние (обобщенные!) очереди
    private Queue<Package> _queueHigh   = new Queue<Package>();
    private Queue<Package> _queueMedium = new Queue<Package>();
    private Queue<Package> _queueLow    = new Queue<Package>();
    //Enqueue - на основании приоритета Package добавляем его
    // в соответствующую очередь
    public void Enqueue(Package item)
    {
        switch(item.Priority) // Package имеет это свойство
        {
        case Priority.High:
          _queueHigh.Enqueue(item);
          break;
        case Priority.Low:
          _queueLow.Enqueue(item);
          break;
        case Priority.Medium:
          _queueMedium.Enqueue(item);
          break;
        }
    }
    // и так далее ...
```

Написание необобщенного класса упрощает тестирование его логики. Затем после тестирования и исправления всех ошибок вы можете сделать контекстную замену `Package` на `<T>` (конечно, все не так прямолинейно, но и не очень отличается от сказанного).

И наконец — обобщенная очередь с приоритетами

Почему очередь с приоритетами рассматривается последней? Это может показаться возвратом назад, ведь вы уже видели код, который использует приоритетную очередь для выполнения задач. Теперь пришло время изучить класс `PriorityQueue`. В этом разделе показан код, а затем даны пояснения, чтобы вы видели, как решается пара небольших проблем. Будем "есть слона по кусочку".

Внутренние очереди

Класс `PriorityQueue` — оболочка, за которой скрываются три обычных объекта `Queue<T>`, по одному для каждого уровня приоритета. Вот первая часть исходного текста `PriorityQueue`, в которой показаны эти три (обобщенные) внутренние очереди:

```
//PriorityQueue - обобщенный класс очереди с приоритетами;
// типы данных, добавляемых в очередь, *обязаны*
// реализовывать интерфейс IPrioritizable
class PriorityQueue<T>
        where T : IPrioritizable
```

```
{
    //Queues - три внутренние (обобщенные!) очереди
    private Queue<T> _queueHigh   = new Queue<T>();
    private Queue<T> _queueMedium = new Queue<T>();
    private Queue<T> _queueLow    = new Queue<T>();
    // Все остальное мы вот-вот рассмотрим...
```

В "полужирных" строках объявляются три закрытых члена-данных типа `Queue<T>`, инициализируемых путем создания соответствующих объектов `Queue<T>`. В разделе "Незавершенные дела" этой главы мы рассмотрим странно выглядящую строку над объявлениями "подочередей".

Метод Enqueue()

Метод `Enqueue()` добавляет элемент типа `T` в `PriorityQueue`. Работа состоит в том, чтобы выяснить приоритет элемента и поместить его в соответствующую приоритету очередь. В первой строке метод получает приоритет элемента и использует конструкцию `switch` для определения целевой очереди исходя из полученного значения. Например, получив элемент с приоритетом `Priority.High`, метод `Enqueue()` помещает его в очередь `queueHigh`. Вот исходный текст метода `PriorityQueue.Enqueue()`:

```
// Добавляет элемент T в очередь на основании значения его
// приоритета
public void Enqueue(T item)
{
    switch (item.Priority) // Требует реализации
    {                      // IPrioritizable
    case Priority.High:
        _queueHigh.Enqueue(item);
        break;
    case Priority.Low:
        _queueLow.Enqueue(item);
        break;
    case Priority.Medium:
        _queueMedium.Enqueue(item);
        break;
    default:
        throw new ArgumentOutOfRangeException(
                item.Priority.ToString(),
                "Неверный приоритет в PriorityQueue.Enqueue");
    }
}
```

Метод Dequeue()

Работа метода `Dequeue()` немного сложнее. Он должен найти непустую очередь элементов с наивысшим приоритетом и выбрать из нее первый элемент. Первую часть своей работы — поиск непустой очереди элементов с наивысшим приоритетом — `Dequeue()` делегирует закрытому методу `TopQueue()`,

который будет описан ниже. Затем метод Dequeue() вызывает метод Dequeue() найденной очереди для извлечения из нее возвращаемого объекта. Вот исходный текст метода Dequeue():

```
//Dequeue - извлечение T из очереди с наивысшим
// приоритетом
public T Dequeue()
{
    // Ищем очередь с наивысшим приоритетом
    Queue<T> queueTop = TopQueue();
    // Очередь не пуста
    if (queueTop != null && queueTop.Count > 0)
    {
        return queueTop.Dequeue(); // Возвращаем первый
                                   // элемент
    }
    // Если все очереди пусты, возвращаем null (здесь
    // можно сгенерировать исключение)
    return default(T);
}
```

Единственная сложность состоит в том, как поступить, если все внутренние очереди пусты, т.е., по сути, пуста очередь PriorityQueue в целом? Что следует вернуть в этом случае? Представленный метод Dequeue() в этом случае возвращает значение null. Таким образом, клиент — код, вызывающий PriorityQueue.Dequeue() — должен проверять, не вернул ли метод Dequeue() значение null. Где именно возвращается значение null? В default(T), в конце исходного текста метода. О выражении default(T) речь пойдет чуть ниже, в разделе "Определение значения null для типа T: default(T)".

Вспомогательный метод TopQueue()

Метод Dequeue() использует вспомогательный метод TopQueue() для того, чтобы найти непустую внутреннюю очередь с наивысшим приоритетом. Метод TopQueue() начинает с очереди _queueHigh и проверяет ее свойство Count. Если оно больше 0, очередь содержит элементы, так что метод TopQueue() возвращает ссылку на эту внутреннюю очередь (тип возвращаемого значения метода TopQueue() — Queue<T>). Если же очередь _queueHigh пуста, метод TopQueue() повторяет свои действия с очередями _queueMedium и _queueLow.

Что происходит, если все внутренние очереди пусты? В этом случае метод TopQueue() мог бы вернуть значение null, но более полезным будет возврат одной из пустых очередей. Когда после этого метод Dequeue() вызовет метод Dequeue() возвращенной очереди, тот вернет значение null. Вот как выглядит исходный текст метода TopQueue():

```
//TopQueue - непустая очередь с наивысшим приоритетом
private Queue<T> TopQueue()
{
    if (_queueHigh.Count > 0)    // Очередь с высоким
      return _queueHigh;         // приоритетом пуста?
    if (_queueMedium.Count > 0) // Очередь со средним
      return _queueMedium;       // приоритетом пуста?
    if (_queueLow.Count > 0)     // Очередь с низким
      return _queueLow;          // приоритетом пуста?
    return _queueLow;            // Все очереди пусты
}
```

Остальные члены PriorityQueue

Полезно знать, пуста ли очередь `PriorityQueue`, и если не пуста, то сколько элементов в ней содержится (каждый объект отвечает сам за себя!). Вернитесь к листингу демонстрационной программы и рассмотрите исходный текст метода `IsEmpty()` и свойства `Count` класса `PriorityQueue`. Может также оказаться полезным включить методы, которые возвращают количество элементов в каждой из внутренних очередей. *Будьте осторожны*: это может слишком многое рассказать о том, как реализована очередь с приоритетами. Держите свою реализацию скрытой.

Использование простого необобщенного класса фабрики

Ранее в главе я использовал объект фабрики для генерации потока объектов типа `Package` со случайными приоритетами. Так как это было давно, повторим его здесь:

```
// PackageFactory - класс, который знает, как создать
// новый пакет Package любого требуемого типа; такой класс
// называется классом-фабрикой.
class PackageFactory
{
    // Генератор случайных чисел.
    Random _randGen = new Random();

    // CreatePackage - метод фабрики, который выбирает
    // случайный приоритет, затем создает пакет с этим
    // приоритетом.
    public Package CreatePackage()
    {
        // Случайным образом выбранный приоритет пакета. Может
        // иметь значение 0, 1 или 2 (значения, меньшие 3).
        int rand = _randGen.Next(3);
        // Используем для генерации нового пакета;
        // приведение типа позволяет использовать значение
        // в конструкции switch.
        return new Package( (Priority) rand);
    }
}
```

Класс `PackageFactory` имеет один член-данные и один метод. (Простую фабрику можно реализовать не как класс, а как метод, например метод в классе `Program`.) При инстанцировании объект `PackageFactory` создает объект класса `Random` и сохраняет его в члене-данных `rand`. `Random` представляет собой библиотечный класс .NET, который генерирует случайные числа.

Использование `PackageFactory`

Для генерации объекта `Package` со случайным приоритетом вызывается метод `CreatePackage()` объекта фабрики:

```
PackageFactory fact = new PackageFactory();
IPrioritizable pack = fact.CreatePackage(); // Обратите
                                  // внимание на интерфейс.
```

Метод `CreatePackage()` запрашивает у своего генератора случайных чисел число от 0 до 2 включительно и использует это число для установки приоритета нового объекта типа `Package`, возвращаемого данным методом (который затем сохраняется в переменной типа `Package` или, еще лучше, в переменной типа `IPrioritizable`).

ЗАПОМНИ!

Обратите внимание на то, что метод `CreatePackage()` возвращает ссылку на `IPrioritizable`, что является более обобщенным решением, чем возврат ссылки на `Package`. Это пример *косвенности* — метод `Main()` обращается к `Package` опосредованно, через интерфейс, который реализует `Package`. Косвенность изолирует метод `Main()` от деталей возвращаемых методом `CreatePackage()` объектов. Это обеспечивает большую свободу в плане изменения реализации фабрики без влияния на метод `Main()`.

Еще немного о фабриках

Фабрики очень удобны для генерации большого количества тестовых данных (фабрика не обязательно использует генератор случайных чисел — он потребовался для конкретной демонстрационной программы `PriorityQueue`). Фабрики усовершенствуют программу, изолируя создание объектов. Каждый раз при упоминании имени определенного класса в вашем исходном тексте вы создаете *зависимость* (dependency) от этого класса. Чем больше таких зависимостей, тем больше степень связности классов, тем "теснее" они связаны между собой.

Программистам давно известно, что следует избегать тесного связывания. (Один из многих методов *развязки* (decoupling) заключается в применении фабрик посредством интерфейсов, как, например, `IPrioritizable`, а не конкретных классов наподобие `Package`.) Программисты постоянно создают объекты

непосредственно, с применением оператора new, и это нормальная практика. Однако использование фабрик может сделать код менее тесно связанным, а следовательно, более гибким.

Незавершенные дела

PriorityQueue все еще нуждается в небольшой доработке.

- Сам по себе класс PriorityQueue не защищен от попыток инстанцирования для типов, например, int, string и Student, т.е. типов, не имеющих приоритетов. Вы должны наложить *ограничения* на класс, чтобы он мог быть инстанцирован только для типов, реализующих интерфейс IPrioritizable. Попытки инстанцировать PriorityQueue для классов, не реализующих IPrioritizable, должны приводить к ошибке времени компиляции.
- Метод Dequeue() класса PriorityQueue возвращает значение null вместо реального объекта. Однако обобщенные типы наподобие <T> не имеют естественного значения null по умолчанию, как, например, int или string. Эта часть метода Dequeue() также требует обобщения.

Добавление ограничений

Класс PriorityQueue должен быть способен запросить у помещаемого в очередь объекта о его приоритете. Для этого все классы, объекты которых могут быть размещены в PriorityQueue, должны реализовывать интерфейс IPrioritizable, как это делает класс Package. Класс Package указывает интерфейс IPrioritizable в заголовке своего объявления:

```
class Package : IPrioritizable
```

После этого он реализует свойство Priority интерфейса IPrioritizable.


ЗАПОМНИ!

Соответствующее ограничение необходимо для PriorityQueue. Нужно, чтобы компилятор немедленно сообщал о проблемах при попытке создать экземпляр для типа, который не реализует IPrioritizable. Компилятор в любом случае сообщит об ошибке, если один из методов обобщенного класса вызовет метод, отсутствующий у типа, для которого инстанцируется обобщенный класс. Однако лучше использовать явные *ограничения*. Поскольку вы можете инстанцировать обобщенный класс буквально для любого типа, должен быть способ указать компилятору, какие типы допустимы, а какие — нет.

ЗАПОМНИ!

Добавить ограничение можно путем указания интерфейса `IPrioritizable` в заголовке `PriorityQueue`:

```
class PriorityQueue<T> where T: IPrioritizable
```

Обратите внимание на выделенную полужирным шрифтом конструкцию, начинающуюся со слова `where`. Это *принудитель* (enforcer), который указывает, что тип `T` обязан реализовывать интерфейс `IPrioritizable`, т.е. как бы говорит компилятору: "Убедись, подставляя конкретный тип вместо `T`, что он реализует интерфейс `IPrioritizable`, а иначе просто сообщи об ошибке".

ЗАПОМНИ!

Вы указываете ограничения, перечисляя в конструкции `where` *одно или несколько* имен:

- имя базового класса, от которого должен быть порожден класс `T` (или должен быть этим классом);
- имя интерфейса, который должен быть реализован классом `T`, как было показано в предыдущем примере;
- прочие ограничения, показанные в табл. 8.1.

Об ограничениях можно прочесть в разделе *Generics, constraints* справочной системы по языку C#.

Таблица 8.1. Варианты обобщенных ограничений

Ограничение	Значение	Пример
`MyBaseClass`	`T` должно быть (или расширять) `MyBaseClass`	`where T: MyBaseClass`
`IMyInterface`	`T` должно реализовывать `IMyInterface`	`where T: IMyInterface`
`struct`	`T` должно быть любым типом-значением	`where T: struct`
`class`	`T` должно быть любым ссылочным типом	`where T: class`
`new()`	`T` должно иметь конструктор без параметров	`where T: new()`

Обратите внимание на ограничения `struct` и `class`. Указание `struct` означает, что `T` может быть любым типом-значением: числовым типом, `char`, `bool` или любым объектом, объявленным с использованием ключевого слова `struct`.

Использование ключевого слова `class` означает, что `T` может быть любым ссылочным типом, т.е. любым *классом*.

Эти ограничения обеспечивают бо́льшую гибкость в достижении требуемого поведения обобщенного класса. А правильное поведение класса превосходит любую цену... Вы не ограничены применением только одного ограничения. Вот пример гипотетического обобщенного класса, объявленного с несколькими ограничениями на `T`:

```
class MyClass<T> : where T: class, IPrioritizable, new()
{ ... }
```

Здесь тип `T` должен быть классом, а не типом-значением; он должен реализовывать интерфейс `IPrioritizable`; а кроме того, он должен иметь конструктор без параметров.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Что если у вас два обобщенных параметра и на оба должны быть наложены ограничения? (Да, вы запросто можете использовать несколько обобщенных параметров — подумайте, например, о `Dictionary<TKey,TValue>`.) Вот как можно использовать две конструкции `where`:

```
class MyClass<T, U> : where T: IPrioritizable, where U: new()
```

Здесь вы видите две конструкции `where`, разделенные запятыми. Первая ограничивает тип `T` объектами, которые реализуют интерфейс `IPrioritizable`. Вторая ограничивает тип `U` объектами, у которых имеется конструктор по умолчанию (без параметров).

Определение значения `null` для типа `T`: `default(T)`

Как упоминалось ранее, у каждого типа есть свое значение по умолчанию, означающее "ничто" для данного типа. Для `int`, `double` и других типов чисел это 0 (или 0.0). Для `bool` это `false`, а для всех ссылочных типов, таких как `Package`, это `null`. Для `string`, как и для прочих ссылочных типов, это значение `null`.

Однако, поскольку обобщенный класс наподобие `PriorityQueue` может быть инстанцирован практически для любого типа данных, C# не в состоянии предсказать, каким должно быть правильное значение `null` в исходном тексте обобщенного класса. Например, в методе `Dequeue()` класса `PriorityQueue` вы можете оказаться именно в такой ситуации: вы вызываете `Dequeue()`, но очередь пуста и пакетов нет. Что вы должны вернуть, что бы могло означать "ничего"? Поскольку `Package` — класс, следует вернуть значение `null`. Это сообщит вызывающему методу, что ничего вернуть не удалось (вызывающий метод, само собой, должен проверять, не вернулось ли значение `null`).

ЗАПОМНИ!

Компилятор не может придать смысл ключевому слову `null` в исходном тексте обобщенного класса, поскольку обобщенный класс может быть инстанцирован для любых типов данных. Вот почему в исходном тексте метода `Dequeue()` используется следующая конструкция:

```
return default(T); // "Правильное" значение null для типа T
```

Эта строка указывает компилятору, что нужно посмотреть, что собой представляет тип `T`, и вернуть верное значение `null` для этого типа. Для `Package`, который в качестве класса представляет собой ссылочный тип, верным возвращаемым значением будет `null`. Однако для некоторых других `T` это значение может быть иным, и компилятор сможет верно определить, что именно следует вернуть.

Пересмотр обобщенности

Модель обобщенности, реализованная в C# 2.0, была неполной. Обобщенность хороша для облегчения жизни программиста, но в этом варианте она мало что делала для облегчения жизни аналитика; с ее помощью было очень трудно смоделировать реальную бизнес-модель. Ситуация изменилась в C# 4.0. Хотя в C# 2.0 все параметры допускают вариантность в нескольких направлениях, это не так в случае обобщенных классов.

Вариантность связана с типами параметров и возвращаемых значений. *Ковариантность* означает, что экземпляр подкласса может использоваться там, где ожидается экземпляр родительского класса, в то время как *контравариантность* означает, что там, где ожидается экземпляр подкласса, может использоваться экземпляр суперкласса. Когда ни то, ни другое невозможно, это называется *инвариантностью*.

Все языки четвертого поколения поддерживают определенную вариантность. В C# 3.0 и более ранних версиях параметры ковариантны, а типы возвращаемых значений — контравариантны. Это работает, потому что строковые и целочисленные параметры ковариантны параметрам-объектам:

```
public static void MessageToYou(object theMessage)
{
    if (theMessage != null)
        Console.Writeline(theMessage);
}

// Затем:
MessageToYou("Это сообщение!");
MessageToYou(4+6.6);
```

А следующий код работает, потому что типы возвращаемых объектов контравариантны строковым и целочисленным возвращаемым типам (например):

```
object theMessage = MethodThatGetsTheMessage();
```

Обобщенные типы являются инвариантными в C# 2.0 и 3.0. Это означает, что типы `Basket<apple>` и `Basket<fruit>` не являются взаимозаменяемыми, как строки и объекты в предыдущем примере.

Вариантность

Если вы посмотрите на метод, подобный следующему

```
public static void WriteMessages()
{
    List<string> someMessages = new List<string>();
    someMessages.Add("Первое сообщение");
    someMessages.Add("Второе сообщение");
    MessagesToYou(someMessages);
}
```

а затем попытаетесь вызвать этот метод так, как поступали ранее в этой главе со строковым типом

```
// Это не работает в C#3!!
public static void MessagesToYou(IEnumerable<object> theMessages)
{
    foreach (var item in theMessages)
        Console.WriteLine(item);
}
```

то это не сработает. Обобщенные типы в C# 3.0 являются инвариантными. Но в Visual Studio 2010 и более поздних версий этот код скомпилируется, поскольку `IEnumerable<T>` является ковариантным: более поздний порожденный тип можно использовать в качестве замены для типа более высокого порядка. С следующем разделе вы увидите конкретный пример.

Контравариантность

Приложение планирования может иметь события `Event`, которые содержат дату, и набор подклассов, одним из которых является `Course`. `Course` — это `Event`. Курсы знают количество своих студентов. Одним из этих их методов является `MakeCalendar`:

```
public void MakeCalendar(IEnumerable<Event> theEvents)
{
    foreach (Event item in theEvents)
    {
        Console.WriteLine(item.WhenItIs.ToString());
    }
}
```

Сделаем вид, что этот код делает календарь. На самом деле пока что все, что он делает, — это выводит дату на консоль. `MakeCalendar` является общесистемным методом, так что он ожидает некоторый перечислимый список событий.

Приложение также имеет алгоритм сортировки с именем `EventSorter`, который передает отсортированную коллекцию в метод `Sort`. Ожидается его вызов из списка событий. Вот класс `EventSorter`:

```
class EventSorter : IComparer<Event>
{
    public int Compare(Event x, Event y)
    {
        return x.WhenItIs.CompareTo(y.WhenItIs);
    }
}
```

Менеджер событий составляет список курсов, сортирует их, а затем составляет календарь. `ScheduleCourses` создает список курсов, а затем вызывает `courses.Sort()` с `EventSorter` в качестве аргумента, как показано далее:

```
public void ScheduleCourses()
{
    List<Course> courses = new List<Course>()
    {
        new Course(){NumberOfStudents=20,
                     WhenItIs = new DateTime(2018,2,1)},
        new Course(){NumberOfStudents=14,
                     WhenItIs = new DateTime(2018,3,1)},
        new Course(){NumberOfStudents=24,
                     WhenItIs = new DateTime(2018,4,1)},
    };
    // Передаем класс ICompare<Event> в коллекцию List<Course>.
    // Это должен быть ICompare<Course>, но можно использовать
    // ICompare<Event> из-за контравариантности.
    courses.Sort(new EventSorter());
    // Передаем список курсов там, где ожидается список событий.
    // Это можно сделать, так как обобщенные параметры ковариантны
    MakeCalendar(courses);
}
```

Но подождите, это же список курсов, который вызывает `Sort()`, а не список событий. Это не имеет значения — `IComparer<Event>` представляет собой контравариантный обобщенный тип для `T` (тип возвращаемого значения) по отношению к `IComparer<Course>`, поэтому все еще можно использовать этот алгоритм.

Вот еще один пример контравариантности, использующий параметры, а не возвращаемые значения. Если у вас есть метод, который возвращает обобщенный список курсов, вы можете вызвать его там, где ожидается список событий, поскольку `Event` является суперклассом для `Course`.

Вы знаете, что у вас может быть метод, который возвращает `String` и присваивает возвращаемое значение переменной, которую вы объявили как объект? Теперь вы можете сделать это и с обобщенной коллекцией.

В общем случае компилятор C# делает предположения о преобразовании обобщенного типа. Пока вы работаете "вверх по цепочке" для параметров или "вниз по цепочке" для возвращаемых типов, C# просто волшебным образом сам определяет соответствующий тип.

Ковариантность

Далее приложение передает список методу `MakeSchedule`, но этот метод ожидает перечислимую коллекцию событий `Event`. Поскольку теперь параметры для обобщенных типов являются ковариантными, можно передать список курсов, так как `Course` является ковариантным для `Event`. Это — проявление ковариантности параметров.

Глава 9

Эти исключительные исключения

В ЭТОЙ ГЛАВЕ...

- »Обработка ошибок с помощью кодов возврата
- » Использование механизма исключений вместо кодов возврата
- » Разработка стратегии обработки исключений

Без сомнения, трудно смириться с тем, что иногда метод не делает то, для чего он предназначался. Это раздражает программистов ничуть не меньше, чем пользователей их программ, тоже часто являющихся источником недоразумений. Вы запрашиваете значение `int`, пользователь вводит число `double`, что приводит к ошибке. Такой метод можно написать так, что он будет просто игнорировать введенный пользователем мусор вместо реального числа, но хороший программист напишет метод таким образом, чтобы он распознавал неверный ввод пользователя и докладывал об ошибке.


ЗАПОМНИ!

В этой главе говорится об ошибках времени выполнения, а не времени компиляции, с которыми C# разберется сам при сборке вашей программы. *Ошибки времени выполнения* происходят не в процессе компиляции, а при выполнении корректно скомпилированной программы.

Механизм исключений C# представляет собой средство для сообщения о таких ошибках способом, который вызывающий метод может лучше понять и использовать для решения возникшей проблемы. Этот механизм имеет массу преимуществ по сравнению с применявшимися ранее методами. В данной главе вы познакомитесь с основами обработки исключений. Здесь вам придется попотеть, так что проверьте, работает ли ваш кондиционер.

Использование механизма исключений для сообщения об ошибках

В C# для перехвата и обработки ошибок используется новый механизм, называемый *исключениями*. Он основан на ключевых словах `try`, `catch`, `throw` и `finally`. Набросать схему его работы можно следующим образом. Метод пытается (`try`) пробраться через кусок кода. Если в нем обнаружена проблема, она бросает[1] (`throw`) индикатор ошибки, который методы могут поймать[2] (`catch`), и независимо от того, что именно произошло, в конце (`finally`) выполнить специальный блок кода, как показано в следующем наброске исходного текста:

```
public class MyClass
{
    public void SomeMethod()
    {
        // Настройка для перехвата ошибки
        try
        {
            // Вызов метода или выполнение каких-то иных
            // действий, которые могут генерировать исключение
            SomeOtherMethod();
            // . . . Какие-то иные действия . . .
        }
        catch(Exception e)
        {
            // Сюда управление передается в случае, если в блоке
            // try сгенерировано исключение – в самом ли блоке, в
            // методе, который в нем вызывается, в методе,
            // который вызывается методом, вызванным в try-блоке,
            // и так далее – словом, где угодно. Объект Exception
            // описывает ошибку
        }
        finally
        {
```

[1] Далее будет использоваться выражение "генерирует исключение". — *Примеч. пер.*

[2] Далее будет использоваться выражение "перехватить исключение". — *Примеч. пер.*

```
            // Выполнение всех завершающих действий: закрытие
            // файлов, освобождение ресурсов и т.п. Этот блок
            // выполняется независимо от того, было ли
            // сгенерировано исключение.
        }
    }

    public void SomeOtherMethod()
    {
        // . . . Ошибка произошла где-то в теле метода . . .
        // . . . И "пузырек" исключения "всплывает" вверх по
        // всей цепочке вызовов, пока не будет перехвачен в
        // блоке catch
        throw new Exception("Описание ошибки");
        // . . . Продолжение метода . . .
    }
}
```

ЗАПОМНИ!

Комбинация `try`, `catch` и, возможно, `finally` называется *обработчиком исключения* (exception handler). Метод `SomeMethod()` помещает некоторую часть кода в блок, помеченный ключевым словом `try`. Любой метод, вызываемый в этом блоке (или метод, вызываемый методом, вызываемым в этом блоке, — и т.д.), рассматривается как вызванный в данном `try`-блоке. Если у вас есть `try`-блок, то должен быть либо блок `catch`, либо блок `finally`, либо оба блока.

ВНИМАНИЕ!

Переменные, объявленные в блоке `try`, `catch` или `finally`, извне блока недоступны. Чтобы получить доступ к переменным в этих блоках, их придется объявлять вне блока:

```
int aVariable;  // Объявление aVariable вне блока.
try
{
    aVariable = 1;
    // Объявление aString в блоке.
    string aString = aVariable.ToString();  // Использование
}                                           // aVariable в блоке.
// Здесь aVariable видима, в отличие от aString.
```

О `try`-блоках

Думайте об использовании `try`-блока как о приведении кода C# в состояние готовности. Если при выполнении какого-либо кода в этом блоке возникнет ошибка, среда выполнения C# сгенерирует исключение. Исключения "всплывают" в коде по вызванным функциям, пока не встретится блок `catch` или приложение не завершится. Блок `try` содержит не только записанные в нем строки кода, но и все методы, вызываемые его содержимым.

О catch-блоках

Обычно непосредственно за блоком try следует ключевое слово catch с блоком, которому передается управление в случае, если где-то в try-блоке произошла ошибка (генерация исключения). Аргумент catch-блока — объект класса Exception (или, чаще, некоторого подкласса класса Exception):

```
catch(Exception e)
{
    // Вывод сообщения об ошибке
    Console.WriteLine(e.ToString());
}
```

Если вам не нужен доступ к информации из объекта перехваченного исключения, вы можете указать в блоке только тип исключения:

```
catch(MyException)
{
    // Действия, которые не требуют обращения к объекту
    // исключения
}
```

Вообще говоря, catch-блок не обязан иметь аргументы: пустой catch перехватывает все исключения, как и catch(Exception):

```
catch
{
}
```

О finally-блоках

Блок finally — если таковой имеется в вашем исходном тексте — выполняется даже в случае перехвата исключения, не говоря уже о том, что он выполняется при нормальной работе. Обычно он предназначается для "уборки" — закрытия открытых файлов, освобождения ресурсов и т.п. Наиболее распространенное применение finally — освобождение ресурсов после кода try-блока, независимо от того, произошло исключение или нет. Поэтому часто можно встретить следующий код:

```
try
{
    ...
}
finally
{
    // Код освобождения ресурсов, такой, например,
    // как закрытие открытых в try-блоке файлов.
}
```

Вы можете использовать блоки `finally` как угодно. Но для каждого блока `try` может существовать только один блок `finally`.


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Метод может иметь несколько обработчиков `try/catch`. Можно даже вложить `try/catch` в `try`-блок, в `catch`-блок или в `finally`-блок (или во все одновременно). То же самое относится и к конструкции `try/finally`.

Что происходит при генерации исключения

При генерации исключения мы получим тот или иной вариант последовательности событий.

1. **Генерируется исключение.**

   Итак, где-то в дебрях кода на неизвестно каком уровне вызовов в методе `SomeOtherMethod()` случилась ошибка... Функция сообщает об этом, генерируя исключение в виде объекта `Exception`, и передает его с помощью оператора `throw` вверх по цепочке вызовов в первый же блок, который в состоянии его перехватить и обработать.

   Заметим еще раз, что исключения представляют собой *ошибки времени выполнения*, а не *времени компиляции*. Это означает, что они происходят во время работы программы. Это может произойти и после того, как вы уже передадите готовую программу заказчику.

2. **C# сворачивает стек вызовов в поисках `catch`-блока.**

   Исключение идет назад к вызывающему методу, затем — к методу, который его вызвал, и так далее вплоть до метода `Main()` программы, пока не будет найден `catch`-блок, способный перехватить исключение. На рис. 9.1 показан путь поиска языком C# обработчика сгенерированного исключения.

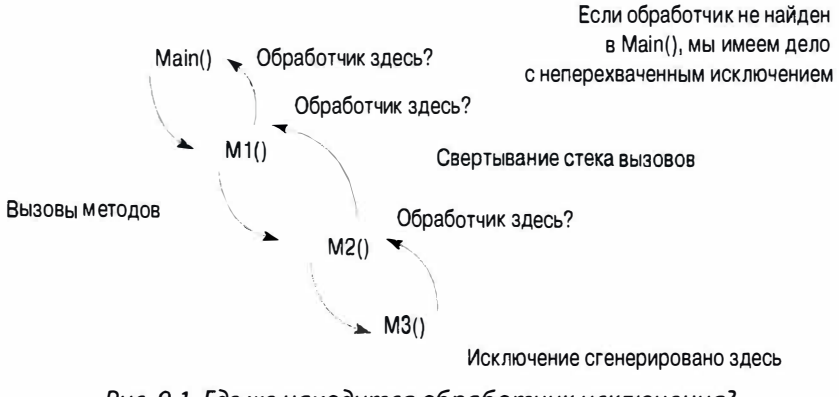


Рис. 9.1. Где же находится обработчик исключения?

3. **Если найден соответствующий блок `catch`, он выполняется.**

   "Соответствующий" `catch`-блок — это блок, который перехватывает класс исключения (или любой из его базовых классов). Этот `catch`-блок может выполнять любое количество действий. Если некоторый метод обладает недостаточным контекстом — т.е. не обладает достаточной для корректной обработки исключения информацией, — он просто не предоставляет `catch`-блок для этого исключения. Подходящий `catch`-блок может оказаться существенно выше в стеке вызовов, чем место генерации исключения.


ЗАПОМНИ!

Механизм исключения превосходит старый механизм возврата кода ошибки.

- Когда вызывающий метод получает код ошибки и не в состоянии корректно ее обработать, он должен явно вернуть ошибку *вызвавшему его* методу, и так далее по цепочке вызовов. Если метод, способный обработать ошибку, находится высоко в стеке, мы получим достаточно уродливую конструкцию.
- Исключение в случае генерации автоматически передается вверх по цепочке вызовов, пока не будет найден его обработчик. Вам никак не надо заботиться о передаче ошибки по стеку, что позволяет не уродовать код.

4. **Если у `try`-блока имеется `finally`-блок, он выполняется независимо от того, было сгенерировано исключение или нет.**

   Код `finally` вызывается перед свертыванием стека к следующему методу в цепочке вызовов. Выполняются *все* блоки `finally` вдоль всей цепочки вызовов.

5. **Если блок `catch` нигде не найден, происходит аварийный останов программы.**

   Если C# возвращается в метод `Main()` и нигде не находит блока `catch`, пользователь получает сообщение о "необработанном исключении", и работа программы на этом аварийно завершается. Однако с неперехваченными исключениями можно справиться при помощи обработчика исключения в методе `Main()` (об этом будет рассказано немного позже в этой главе).

Этот механизм более сложен в работе и труден для понимания, чем применение кодов ошибок. Однако сопоставьте возросшую сложность со следующими соображениями.

- Исключения предоставляют более "выразительную" модель, которая позволяет выразить большое количество стратегий обработки ошибок.
- Объект исключения содержит гораздо больше информации (что особенно важно при отладке), чем коды ошибок.

- Исключения приводят к меньшему количеству кода, причем к коду более удобочитаемому.
- Исключения являются неотъемлемой частью C#, в отличие от узкоспециализированных схем кодов ошибок, любые две из которых существенно отличаются одна от другой.
- Согласованность модели обработки ошибок облегчает понимание.

Генерация исключений

Если классы из библиотеки .NET могут генерировать исключения, то и вы можете делать это. Для генерации исключения при обнаружении ошибки воспользуйтесь ключевым словом `throw`:

```
throw new ArgumentException("Не спорь со мной!");
```

У вас столько же прав на генерацию исключений, сколько и у любого другого. Поскольку библиотека классов .NET понятия не имеет о вашем собственном `BadHairDayException`, кто, кроме вас, может сгенерировать такое исключение?

СОВЕТ

Если ваша ситуация описывается одним из предопределенных исключений .NET, сгенерируйте его. Но если не подходит ни одно из них, вы можете создать собственный *пользовательский класс исключения*.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В .NET имеется несколько типов исключений, которые вы никогда не должны генерировать: `StackOverflowException`, `OutOfMemoryException`, `ExecutionEngineException` и еще несколько типов, связанных с кодом, не являющимся кодом .NET. Это системные исключения. Например, если у вас недостаточно места в стеке (`StackOverflowException`), у вас просто нет памяти для продолжения выполнения программы. Учитывая, что обработка исключений происходит в стеке, у вас даже недостаточно памяти для продолжения обработки исключения. Аналогично `OutOfMemoryException` определяет условие, в котором ваше приложение исчерпало память кучи (которая используется для ссылочных переменных). И если механизм исключений вообще не работает (`ExecutionEngineException`), то нет никакого смысла продолжать работу, потому что у вас нет никакого способа обработать эту ошибку.

Для чего нужны исключения


ЗАПОМНИ!

Программа, которая не в состоянии решить, что делать в определенной ситуации, должна сгенерировать исключение. Если, например, предполагается, что метод должен обработать весь массив или прочесть весь файл, но по каким-то причинам не в состоянии это сделать, он должен сгенерировать соответствующее исключение.

Метод может не справиться со своей задачей по многим причинам: некорректные входные данные, неожиданные условия (например, отсутствующий файл или файл меньшего, чем следует, размера) и т.д. Задача оказывается незавершенной или в принципе невыполнимой. В таком случае следует сгенерировать исключение.


ЗАПОМНИ!

Основная идея такова: кто бы ни вызвал этот метод, он должен знать, что задача не завершена. Генерация исключения почти всегда лучше возврата кода ошибки. Однако если вы можете обработать исключение внутри метода, вам нужно сделать это, а не использовать исключение вместо написания хорошего кода. Дополнительные идеи о корректном применении исключений можно почерпнуть из статьи по адресу `https://docs.microsoft.com/dotnet/standard/exceptions/best-practices-for-exceptions`.

Исключительный пример

В демонстрационной программе `FactorialException` приведены ключевые элементы механизма исключений.

```
// FactorialException - создание программы вычисления
// факториала, которая сообщает о некорректном аргументе с
// использованием исключений
using System;

namespace FactorialException
{
    // MyMathFunctions - набор созданных мною
    // математических функций
    public class MyMathFunctions
    {
        // Factorial - возвращает факториал переданного
        // аргумента
        public static int Factorial(int value)
        {
            // Проверка: отрицательные значения запрещены
```

```
        if (value < 0)
        {
            // Сообщение об отрицательном аргументе
            string s = String.Format(
                "Отрицательный аргумент в вызове Factorial {0}",
                value);
            throw new ArgumentException(s);
        }

        // Начинаем со значения аккумулятора,
        // равного 1
        int factorial = 1;

        // Цикл со счетчиком, уменьшающимся до 1, с умножением
        // на каждой итерации значения аккумулятора на
        // величину счетчика
        do
        {
            factorial *= value;
        }
        while (--value > 1);

        // Возвращаем вычисленное значение
        return factorial;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Обработчик ошибки.
        try
        {
            // Вызов функции вычисления факториала в
            // цикле от 6 до -6
            for (int i = 6; i > -6; i--)
            {
                // Вычисление факториала.
                int factorial = MyMathFunctions.Factorial(i);
                // Вывод результата на каждой итерации
                Console.WriteLine("i = {0}, факториал = {1}",
                                  i, factorial);
            }
        }
        catch (ArgumentException e)
        {
            // Это обработчик исключения "в последний момент" -
            // на уровне метода Main(). Пожалуй, все, что вы
            // можете сделать, - это сообщить о случившемся
            // пользователю.
            Console.WriteLine("Фатальная ошибка:");
            // При выпуске окончательной версии программы
```

```
            // желательно заменить этот код пояснением на
            // простом языке, понятном пользователю, что же
            // именно произошло и что делать в такой ситуации.
            Console.WriteLine(e.ToString());
        }

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
  }
}
```

Эта "исключительная" версия функции `Main()` практически полностью находится в `try`-блоке. Блок `catch` в конце функции `Main()` перехватывает объект `ArgumentException` и использует его метод `ToString()` для вывода информации об ошибке, содержащейся в этом объекте в виде строки. В примере использован класс `ArgumentException`, поскольку это исключение наиболее точно описывает ситуацию: неприемлемый аргумент метода `Factorial()`.

Что делает этот пример "исключительным"

Этот метод `Factorial()` включает проверку на отрицательность переданного аргумента. Если аргумент отрицателен, метод `Factorial()` не может продолжать работу; он формирует сообщение об ошибке с описанием ситуации, включая само отрицательное значение, вызвавшее ошибку. Затем метод `Factorial()` вносит информацию во вновь создаваемый объект типа `ArgumentException`, который передается с помощью механизма исключений вызывающей функции.

СОВЕТ

Вы можете использовать отладчик, чтобы наблюдать работу исключения. Вывод этой программы выглядит следующим образом:

```
i = 6, факториал = 720
i = 5, факториал = 120
i = 4, факториал = 24
i = 3, факториал = 6
i = 2, факториал = 2
i = 1, факториал = 1
i = 0, факториал = 0
Фатальная ошибка:
System.Exception: Отрицательный аргумент в вызове Factorial -1
at Factorial(Int32 nValue) in
    c:\c#programs\Factorial\Program.cs:line 23
at FactorialException.Program.Main(String[] args) in
    c:\c#programs\Factorial\Program.cs:line 56
Нажмите <Enter> для завершения программы...
```

В первых нескольких строках выводятся корректно вычисленные факториалы чисел от 6 до 0. Попытка вычислить факториал для –1 приводит к генерации исключения.

В первой строке сообщения об ошибке выводится информация, сгенерированная в функции `Factorial()`. Эта строка описывает природу ошибки, включая вызвавшее неприятности значение аргумента –1.

Трассировка стека

В оставшейся части вывода выполняется *трассировка стека*. В первой строке указывается, в какой функции сгенерировано исключение. В данном случае это было сделано в функции `Factorial(int)`, а именно — в строке 23 исходного файла `Program.cs`. Функция `Factorial()` была вызвана из функции `Main(string[])` в строке 56 того же файла. На этом трассировка файла прекращается, поскольку функция `Main()` содержит блок, перехвативший и обработавший указанное исключение.

Вы должны согласиться, что это весьма впечатляюще. Сообщение об ошибке описывает случившееся и позволяет указать аргумент, приведший к ней. Трассировка стека полностью отслеживает, где именно и в результате какой последовательности вызовов произошла ошибка. При такой диагностике поиск ошибки и ее причины не должны составить никакого труда.


СОВЕТ

Выполнив предыдущий пример и просмотрев содержимое стека вызовов, выведенное на консоль, вы увидите метод `Main()` в *нижней* части листинга. Я же обычно предпочитаю рассматривать вызывающие функции как находящиеся над вызываемыми, как было показано на рис. 9.1.


СОВЕТ

Возврат дополнительной информации наподобие стека вызовов очень полезен при разработке приложения, но вряд ли стоит показывать его пользователям в конечной версии. Тем не менее даже в этом случае его можно записывать в журнальный файл.


СОВЕТ

Пока программа работает в отладчике, трассировка стека доступна в одном из окон отладчика Visual Studio.

Использование нескольких `catch`-блоков

Как упоминалось ранее, вы можете определить собственные типы исключений. Предположим, вы определили класс `CustomException`. Этот класс может иметь, например, следующий вид:

```
public class CustomException : System.Exception
{
    // Конструктор по умолчанию
    public CustomException() : base()
    {
    }
    // Конструктор с аргументом
    public CustomException(String message):base(message)
    {
    }
    // Конструктор с аргументом и с исключением
    public CustomException(String message,
                           Exception innerException)
    :base(message, innerException)
    {
    }
    // Конструктор с аргументом и поддержкой сериализации
    protected CustomException(SerializationInfo info,
                              StreamingContext context):
    base(info, context)
    {
    }
}
```

Вы можете использовать эту схему в качестве основы для любого пользовательского исключения, которое вы хотите создать. Здесь нет никакого специального кода (если только вы не захотите его добавить); записи `base()` означают, что этот код использует код из `System.Exception`. То, что вы видите здесь, — это проявление наследования. Другими словами, пока что вам не нужно слишком беспокоиться о том, как работает это пользовательское исключение.

Теперь рассмотрим конструкцию `catch`.

```
public void SomeMethod()
{
    try
    {
        SomeOtherMethod();
    }
    catch(CustomException ce)
    {
    }
}
```

Что произойдет, если `SomeOtherMethod()` сгенерирует простое исключение `Exception` или какое-то другое исключение, отличное от `CustomException`? Это будет выглядеть как игра в футбол бейсбольным мячом: ворота не будут соответствовать мячу.

ЗАПОМНИ!

К счастью, C# позволяет программе определять несколько конструкций `catch`, каждая из которых предназначена для своего типа исключения. Вы можете использовать их одну за другой, выстроив несколько конструкций `catch` для разных типов исключений одну за другой после блока `try`. C# последовательно проверяет каждый блок `catch`, сравнивая сгенерированное исключение с типом аргумента `catch`, как показано в следующем фрагменте кода:

```
public void SomeMethod()
{
    try
    {
        SomeOtherMethod();
    }
    catch(CustomException ce) // Наиболее конкретный тип
    {
        // Все объекты CustomException будут перехвачены
        // в этом блоке
    }
    // Здесь можно добавить блоки для других исключений
    catch(Exception e)    // Наиболее общий тип исключения
    {
        // Все неперехваченные к этому моменту исключения
        // перехватываются здесь.
    }
}
```

Если метод `SomeOtherMethod()` сгенерирует объект `Exception`, он пройдет через `catch(CustomException)`, так как `Exception` не является типом `CustomException`. Это исключение будет перехвачено следующей конструкцией `catch`, а именно — `catch(Exception)`.

ВНИМАНИЕ!

Всегда располагайте `catch`-блоки от наиболее специализированного к наиболее общему. Никогда не размещайте более общий блок первым, как это сделано в приведенном фрагменте исходного текста:

```
public void SomeMethod()
{
    try
    {
        SomeOtherMethod();
    }
    catch(Exception me) // Самый общий блок - это неверно!
    {
        // Все объекты CustomException будут перехвачены здесь
    }
    catch(CustomException e)
    {
        // Сюда не доберется ни один объект - все они будут
        // перехвачены более общим блоком
    }
}
```

Более общий блок отнимает объекты исключений у более специализированного блока. К счастью, компилятор в состоянии обнаружить такую ошибку и предупредить о ее наличии.

ЗАПОМНИ!

Любой класс, наследующий `CustomException`, ЯВЛЯЕТСЯ `CustomException`:

```
class MySpecialException : CustomException
{
    // ... что-то там ...
}
```

Блок `catch` для типа `CustomException` перехватит объект `MySpecialException`, как лягушка муху...

Планирование стратегии обработки ошибок

При разработке программы имеет смысл заранее иметь точный план обработки ошибок. Использование исключений вместо кодов ошибок — это только первое решение, но далеко не последнее.

Вопросы, помогающие при планировании

При разработке программ следует все время держать в памяти некоторые важные вопросы.

- **Что может пойти не так?** Спрашивайте себя об этом при работе над каждым фрагментом кода.
- **Если что-то идет не так, могу я исправить ситуацию?** Если да, то вы можете восстановить нормальное состояние программы и продолжить ее работу. Если нет, то, пожалуй, вам следует паковать чемоданы...
- **Подвергаются ли данные пользователя риску?** Если да, вы должны сделать все, что в ваших силах, для предотвращения потери или повреждения данных. Осознанно выпускать программу, которая в состоянии повредить пользовательские данные, — преступная халатность.
- **Где следует разместить обработчик исключения для данной ситуации?** Попытка обработать исключение в методе, в котором оно сгенерировано, — не всегда лучшее решение. Часто некоторый другой метод в цепочке вызовов имеет больше информации и в состоянии более интеллектуально и эффективно справиться с возникшей ситуацией. Размещайте блоки `try/catch` так, чтобы блок `try` охватывал вызовы, в которых возможна генерация исключений.

- **Какие исключения я должен обрабатывать?** Перехватывайте все исключения, для которых вы в состоянии восстановить нормальное состояние. Обязательно попытайтесь найти способ восстановления. В процессе разработки и тестирования необработанные исключения будут достигать верхушки вашей программы. Перед тем как передать программу пользователям, исправьте все случаи возможного возникновения необработанного исключения. Однако иногда исключение *должно* потребовать завершения программы, когда ситуация безнадежна.
- **Как быть с исключениями, которые проскакивают через мою защиту?** В разделе "Последний шанс перехвата исключения" ниже в этой главе рассказывается, как обеспечить перехват таких "паршивых овец".
- **Насколько надежным (безаварийным) должен быть мой код?** Если ваш код работает в системе управления движением воздушного транспорта, он должен быть очень надежен. Если это утилита на один запуск, можно несколько расслабиться...

Советы по написанию кода с хорошей обработкой ошибок

Вы должны все время помнить о вопросах из предыдущего раздела при работе над программами. Кроме того, вам могут помочь следующие советы.

- **Любой ценой защищайте пользовательские данные.** Это самое главное. См. также следующий пункт.
- **Избегайте аварийного останова программы.** Если это возможно, постарайтесь восстановить функционирование программы, если нет, завершите ее как можно более "мягко". Не позволяйте вашей программе вдруг выдать пользователю нечто невразумительное и прекратить работу. Под *мягко* имеется в виду предоставление ясных сообщений о том, что произошло и как этого избежать в другой раз, а также закрытие программы с корректным освобождением ресурсов и сохранением пользовательских данных. Пользователи ненавидят внезапные аварийные остановы программ.
- **Не позволяйте программе работать, если вы не можете восстановить ее нормальное состояние.** Программа может оказаться в нестабильном состоянии, а пользовательские данные — несогласованными. Если возможности корректной обработки ситуации нет, выводите соответствующее сообщение и немедленно завершайте программу вызовом `System.Environment.FailFast()`. Это не авария, а спланированная остановка.

- **Рассматривайте библиотеки классов и приложения по-разному.** В *библиотеках классов* позволяйте исключениям достигать вызывающего метода, который лучше знает, как справиться с возникшей проблемой. Не оставляйте вызывающий метод в неведении относительно того, что произошло. Но в *приложении* обрабатывайте все исключения, какие можете. Главное — чтобы код оставался как можно дольше работоспособным и защищал пользовательские данные без множества несущественных сообщений.
- **Генерируйте исключения, если по какой-то причине метод не в состоянии завершить выполнение своей задачи.** Вызывающий метод должен знать о том, что возникла проблема (это может быть метод выше в стеке вызовов или некоторый метод в коде, написанном другим программистом и использующим ваш код). Если при проверке корректности входных данных перед их использованием выясняется их непригодность — например, обнаруживается неожиданное значение `null`, — по возможности исправьте ситуацию и продолжайте работу; в противном случае сгенерируйте исключение.
- **Пытайтесь писать код, который не требует генерации исключений**, исправляйте ошибки при их обнаружении, не полагаясь на исключения. Но при необходимости сообщения об ошибках и их обработке как главный метод используйте исключения.
- **Старайтесь не перехватывать исключения там, где вы не в состоянии обработать их максимально корректно, предпочтительно — с восстановлением работоспособности программы.** Перехват исключения, которое вы не в состоянии обработать, напоминает ловлю осы голой рукой. Большинство методов не содержат обработчики исключений.
- **Тщательно тестируйте свой код, в особенности для некорректных входных данных.** Может ли ваш метод работать с отрицательными входными данными? С нулем? С очень большими числами? С пустой строкой? Со значением `null`? Что еще может сотворить пользователь, чтобы вызвать исключение? Какие ресурсы, способные привести к ошибкам, использует ваш код? Файлы, базы данных, URL? (См. два предыдущих пункта.)
- **Перехватывайте как можно более конкретные исключения.** Не пишите слишком много `catch`-блоков для высокоуровневых классов исключений типа `Exception` или `ApplicationException`. Вы рискуете не пропустить такие исключения вверх по цепочке вызовов.
- **Всегда размещайте обработчик "последнего шанса" в методе `Main()` — или в ином месте "на вершине" программы (за исключением библиотек классов).** В таком блоке можно перехватить исключение `Exception`. Перехватывайте и обрабатывайте все

исключения по ходу работы программы, предоставив блоку "последнего шанса" перехват "отстающих". (См. раздел "Последний шанс перехвата исключения" далее в этой главе.)

- **Не используйте исключения как часть обычного потока выполнения.** Например, не надо использовать исключения как способ выхода из цикла или из метода.
- **Подумайте о создании собственных классов исключений**, если они должны нести с собой дополнительную информацию, которая может помочь в отладке или предоставить пользователю более осмысленное сообщение об ошибке.

Оставшаяся часть этой главы дает вам в руки инструменты, необходимые для следования приведенным советам. Еще одним источником информации является раздел *exception handling, design guidelines* справочной системы языка C#.


ЗАПОМНИ!

Если открытые методы генерируют исключения, которые вызывающему методу может потребоваться перехватить, то такие исключения являются частью открытого интерфейса класса. Вы должны их документировать, предпочтительно при помощи комментариев для XML-документации.

Анализ возможных исключений метода

Рассмотрим следующий метод. Какие исключения он может генерировать? Ответ на этот вопрос — первый шаг в создании обработчиков исключений.

```
public string FixNamespaceLine(string line)
{
    const string COMPANY_PREFIX = "CMSCo";
    int spaceIndex = line.IndexOf(' ');
    int nameStart =
            GetNameStartAfterNamespaceKeyword(line, spaceIndex);
    string newline = string.Empty;
    newline =
            PlugInNamespaceCompanyQualifier(line, COMPANY_PREFIX,
                                            nameStart);
    return newline.Trim();
}
```

Этот метод представляет собой часть некоторого кода, предназначенного для поиска ключевого слова `namespace` в файле и вставки строки, представляющей имя компании, в качестве префикса имени пространства имен. Следующий пример демонстрирует, где обычно находится в файле C# ключевое слово `namespace`.

```
using System;
namespace SomeName
{
    // Код в пространстве имен...
}
```

В результате вызова метода `FixNamespaceLine()` в файле с исходным текстом показанная далее первая строка превратится во вторую:

```
namespace SomeName
namespace CmsCo.SomeName
```

Полная программа читает `.CS`-файлы. Затем она проходит их строка за строкой, применяя к каждой метод `FixNamespaceLine()`. Для переданной строки кода метод вызывает `String.IndexOf()` для поиска индекса имени пространства имен (обычно 10). Затем он вызывает `GetNameStartAfterNamespaceKeyword()`, чтобы найти начало имени пространства имен. Наконец вызывается еще один метод, `PlugInNamespaceCompanyQualifier()`, для вставки имени компании в требуемое место строки, которая затем будет возвращена методом. Основная часть работы выполняется методами-субподрядчиками.

Начнем с того, что, даже не зная предназначение метода или того, что и как делают два вызываемых в нем метода, рассмотрим входные данные. Аргумент `line` может привести как минимум к одной проблеме при вызове `String.IndexOf()`. Если значение `line` равно `null`, вызов `IndexOf()` генерирует исключение `ArgumentNullException`. Метод нельзя вызывать для объекта `null`. Кроме того, будет ли работать вызов `IndexOf()` для пустой строки? Оказывается, будет, но что случится при передаче пустой строки `line` одному из методов с длинными именами? Я бы рекомендовал добавить проверку в первую строку метода `FixNamespaceLine()` и проверить значение аргумента как минимум на равенство `null`:

```
if(String.IsNullOrEmpty(name))  // Очень удобный метод.
{
    return name; // Вместо генерации исключения можно вернуть
                 // полученную строку назад, что вполне логично.
}
```

Далее, после того как вы успешно миновали вызов `IndexOf()`, один из двух вызовов методов может сгенерировать исключение, даже при предварительной проверке `line`. Если `spaceIndex` вернет –1 (подстрока не найдена) — что вполне возможно в большинстве случаев, так как передаваемая строка обычно не содержит ключевого слова `namespace`, — передача ее первому методу может вызвать проблемы. Следует защитить и этот код:

```
if (spaceIndex > -1) ...
```

Если значение `spaceIndex` отрицательно, ключевого слова `namespace` в строке нет. *Это не ошибка.* Эту строку надо просто пропустить и вернуть ее неизменной, после чего перейти к следующей строке. В любом случае прочие методы вызывать не надо.

Вызовы методов требуют дополнительного изучения вопроса о том, какие исключения каждый из них может генерировать, затем надо рассмотреть все вызываемые из них методы и так далее, пока не доберемся до окончания этой цепочки.

Где же с учетом всего сказанного следует поместить обработчик исключений?

Вы можете захотеть поместить бо́льшую часть тела метода `FixNamespaceLine()` в `try`-блок. Но насколько хорошее это решение? Это — низкоуровневый метод, так что при необходимости он должен генерировать собственные исключения или передавать дальше исключения, сгенерированные в вызываемых им методах. Я бы рекомендовал просмотреть всю цепочку вызовов, чтобы выяснить, какой же метод лучше всего подходит для размещения обработчика.

При перемещении по цепочке вызовов постоянно задавайте сами себе вопросы из раздела "Вопросы, помогающие при планировании". Что произойдет, если `FixNamespaceLine()` сгенерирует исключение? Это зависит от того, каким образом результат работы этого метода используется методами, находящимися выше в цепочке. Кроме того, какими неприятностями это чревато? Если вы не сможете "исправить" строки с пространствами имен, будет ли пользователем потеряно что-то важное? Можно ли оставить файл неисправленным (в этом случае можно перехватить исключение пониже в цепочке и сообщить пользователю о необработанном файле)? Думаю, что основную идею вы поняли. Мораль, вытекающая из всего сказанного, проста: создание обработчика исключений требует анализа и раздумий.


ЗАПОМНИ!

Помните, что *любой* вызов метода может привести к генерации исключений, например приложению *может* не хватить памяти или может не оказаться какой-то необходимой для работы приложения сборки или библиотеки. В этом случае вы практически ничего не сможете сделать...

Как выяснить, какие исключения генерируются теми или иными методами


СОВЕТ

Как при вызове некоторого метода из библиотеки классов .NET, такого как `String.IndexOf()` — или даже одного из ваших собственных методов, — узнать, может ли он генерировать исключения?

- **Visual Studio предоставляет помощь в виде всплывающих подсказок.** Если вы наведете указатель мыши на имя метода в редакторе Visual Studio, желтая всплывающая подсказка перечислит не только параметры метода и его возвращаемый тип, но и исключения, которые он может генерировать.
- **Если у вас есть XML-комментарии к вашим собственным методам, Visual Studio будет выводить эту информацию в подсказках так же, как и для методов .NET.** Если вы документируете исключения, которые может генерировать ваш метод, то увидите их в подсказке. Поместите информацию об исключениях в раздел `<exception>` в комментарии `<summary>` для ее вывода во всплывающей подсказке.
- **Еще больше информации содержится в справочной системе.** Если вы найдете метод .NET в справочной системе, то увидите список всех исключений, которые может генерировать данный метод, а также дополнительную информацию, которая во всплывающих подсказках не отображается. Чтобы получить справку по методу, щелкните на его имени в коде и нажмите <F1>. Можно также добавить аналогичную справку и для собственных классов и методов.

Следует рассмотреть все перечисленные исключения и решить, какие из них, каким именно образом и где обрабатывать.

Последний шанс перехвата исключения

Программа `FactorialException` помещает все тело метода `Main()` — за исключением последнего вывода на консоль — в обработчик исключений "последнего шанса".


ЗАПОМНИ!

При разработке приложения всегда размещайте содержимое метода `Main()` в `try`-блоке, так как `Main()` — стартовая точка программы, а значит, и конечная точка тоже. (Если вы пишете библиотеку классов, предназначенную для повторного использования, о необработанных исключениях можете не беспокоиться: о них должен позаботиться тот, кто эту библиотеку использует.)

Любое неперехваченное ранее исключение добирается до метода `Main()`. Это последняя возможность перехватить исключение и не дать ему добраться до Windows, когда сообщение об ошибке окажется не слишком вразумительным и не даст конечному пользователю понять, что же произошло и как этого избежать.

В программе FactorialException весь серьезный код метода Main() находится в try-блоке. Связанный с ним catch-блок перехватывает все исключения, выводит сообщение на консоль, и работа приложения завершается.

Этот catch-блок служит для предотвращения аварийного останова программы путем перехвата всех исключений, не обработанных ранее. Это ваш шанс пояснить пользователю, что же произошло и почему приложение закрывается.

Чтобы понять, зачем нужен этот обработчик "последнего шанса", напишите маленькую программу, в которой специально сгенерированное исключение не будет перехвачено, и посмотрите, что видит в этой ситуации конечный пользователь.


ЗАПОМНИ!

В процессе разработки вы, конечно, *хотите* видеть все исключения, генерируемые при тестировании вашего кода, в их, так сказать, естественной среде обитания, так что вас интересует специфическая информация об исключении, месте его генерации и т.п. В конечной версии программы техническую информацию следует преобразовать в обычные слова, понятные каждому пользователю программы. Эти слова по возможности должны включать советы, как не допустить повторения этой ситуации еще раз. Обработчик "последнего шанса" должен вести журнал с записью всей информации об исключениях, включая техническую, для дальнейшего анализа и усовершенствования программы.

Генерирующие исключения выражения

Версии C # до 7.0 имеют определенные ограничения, когда речь заходит о генерации исключения как части выражения. В ранних версиях у вас было два варианта. Первый вариант — завершить выражение, а затем проверить результат, как показано далее:

```
var myStrings = "One,Two,Three".Split(',');
var numbers = (myStrings.Length > 0) ? myStrings : null
if(numbers == null){throw new Exception("Чисел нет!");}
```

Второй вариант — сделать исключение частью выражения, как показано здесь:

```
var numbers = (myStrings.Length > 0) ?
            myStrings :
            new Func<string[]>(() => {
                    throw new Exception("Чисел нет!"); })();
```

C# 7.0 и более поздние версии включают новый оператор — ?? (два вопросительных знака). Вы можете сократить два предыдущих примера до следующего вида:

```
var numbers = myStrings ?? throw new Exception("Чисел нет!");
```

В этом случае, если myStrings равен null, код автоматически генерирует исключение. Вы можете использовать эту методику и в условном операторе (как во втором примере):

```
var numbers = (myStrings.Length > 0)? myStrings :
        throw new Exception("Чисел нет!");
```

Возможность генерировать выражения имеется и для членов с выражениями. Возможно, вы видели такие члены в одной из двух следующих разновидностей (если нет, то вы встретитесь с ними в части 2, "Объектно-ориентированное программирование на C#", так что можете не беспокоиться о том, что это означает):

```
public string getMyString()
{
    return "One,Two,Three";
}
```

или

```
public string getMyString() => "One,Two,Three";
```

Однако предположим, что вы не знаете, какое содержимое следует предоставить. В этом случае до версии 7.0 выбор был небогат:

```
public string getMyString() => return null;
```

или

```
public string getMyString() {throw NotImplementedException();}
```

Оба варианта не без проблем. В первом случае вызывающая программа остается без представления о том, произошел ли сбой в методе: ведь нулевое возвращаемое значение может быть ожидаемым значением. Вторая версия слишком громоздка — вам нужно создать стандартную функцию только для того, чтобы вызвать исключение. Благодаря новым дополнениям к C# 7.0 и более поздним теперь имеется возможность генерировать выражения, и предыдущие строки превращаются в

```
public string getMyString() => throw new NotImplementedException();
```

Глава 10

Списки элементов с использованием перечислений

В ЭТОЙ ГЛАВЕ...

- Примеры перечислений в реальном мире
- Создание и применение перечислений
- Использование перечислений для определения флагов
- Использование перечислений как частей инструкций выбора

Перечислять означает "указывать отдельные элементы как располагающиеся в списке". Например, вы можете создать перечисление цветов, а затем перечислить отдельные цвета, такие как красный, синий, зеленый и т.д. Использовать перечисления в программировании имеет смысл постольку, поскольку вы можете перечислять отдельные элементы как часть общей коллекции. Например, `Colors.Blue` будет обозначать синий цвет, а `Colors.Red` — красный. В силу удобства перечислений они широко используются в реальном мире, а потому вы видите их и в приложениях. Код должен моделировать реальный мир, чтобы обеспечить полезную функциональность в удобной для понимания форме.

Создавать перечисления в C# позволяет ключевое слово `enum`. Эта глава начинается с обсуждения основных применений перечислений, а затем речь идет о некоторых интересных дополнениях. Например, для определения начального значения каждого элемента перечисления можно использовать *инициализаторы*.

Флаги предоставляют компактный способ отслеживания небольших параметров конфигурации — обычно они включены или выключены, но их можно сделать и более сложными. Флаги часто используются в старых приложениях, потому что делают использование памяти значительно более эффективным. Приложения C# используют флаги для группирования схожих настроек и упрощения их поиска и работы с ними. Можно использовать одну переменную-флаг для точного определения, как должны работать некоторые объекты.

Перечисления также используются в конструкциях `switch`. С ними вы познакомились в главе 5, "Управление потоком выполнения", а в этой главе мы пойдем немного дальше: вы узнаете, как использование перечислений может сделать ваши конструкции `switch` еще проще для чтения и понимания.

Перечисления в реальном мире

Многие программные конструкции связаны с реальным миром, и это вполне относится и к перечислениям. Перечисление — это любая постоянная коллекция предметов. Как упоминалось ранее, цвета являются одним из наиболее распространенных перечислений, и вы часто используете их в реальном мире. Однако, если бы вы искали перечисления цветов в Интернете, то обнаружили бы массу ссылок. Вместо перечисления цветов ищите "цветовой круг"; обычно именно так люди в реальном мире перечисляют цвета и создают наборы цветовых типов (описание цветового круга и калькулятора цветов см. по адресу `https://www.sessions.edu/color-calculator/`).

Коллекции принимают разные формы, и вы можете даже не осознавать, что создали одну из них. Например, на сайте `http://www.softschools.com/science/biology/classification_of_living_things/` рассказывается о классификации живых организмов. Поскольку эти классификации следуют определенному шаблону и, как правило, не сильно меняются, вы можете выразить их в приложении как перечисление. Например, вряд ли когда-либо изменится список пяти царств[1]. Даже список типов в каждом царстве вряд ли изменится, так что их также можно выразить как перечисления.

Практическое повседневное использование перечислений включает списки предметов или информацию, которая нужна всем. Например, вы не сможете

[1] В 1977 году к царствам животных, растений, грибов, бактерий и вирусов добавлены царства протистов и археев, а в 1998 году — царство хромистов. — *Примеч. пер.*

отправить что-либо по почте, не зная, в какую страну должно быть доставлено ваше отправление. Перечисление государств экономит время и обеспечивает проверку корректности адреса. Перечисления используются для правильного представления реальных объектов. Люди делают ошибки, а перечисления уменьшают их количество. Кроме того, поскольку они экономят время, люди действительно часто к ним прибегают.


ЗАПОМНИ!

Перечисления работают только при определенных условиях. На самом деле зачастую возникают ситуации, в которых вам, определенно, не следует использовать перечисление. В следующем списке приведены некоторые практические правила, которые следует использовать при принятии решения о создании перечисления.

- **Стабильность коллекции**. Коллекция должна представлять стабильный, неизменный список членов. Список государств меняется не так уж часто даже в наше нестабильное время. Список десяти лучших песен в чарте Billboard нестабилен и может меняться почти каждый день.
- **Стабильность членов**. Каждый член коллекции также должен оставаться стабильным и представлять узнаваемую, согласованную ценность. Список кодов городов, хотя и довольно велик, непротиворечив и распознаваем, поэтому при необходимости вы можете создать соответствующее перечисление. Список имен людей является плохой идеей, потому что люди постоянно меняют написание и произношение имен и с легкостью добавляют новые имена.
- **Согласованное значение**. Перечисления обеспечивают связь между числовыми значениями, которые может понять программа, и словами, которые может понять человек. Если числовое значение, связанное с конкретным словом, изменится, перечисление перестанет работать, поскольку вы не сможете полагаться на надежную связь между числовым значением и словом, используемым для его представления.

Работа с перечислениями

Основная идея, лежащая в основе перечислений, относительно проста. Все, что вам действительно нужно сделать, — это создать список имен и присвоить имя получившейся коллекции. Однако вы можете воспользоваться дополнениями к перечислению, которые повысят гибкость и позволят использовать перечисления в широком диапазоне сценариев. В следующих разделах описывается, как создавать различные виды перечислений.

Использование ключевого слова enum

Ключевое слово enum используется при создании перечисления. Например, приведенный далее код создает перечисление с именем Colors.

```
enum Colors {Red, Orange, Yellow, Green, Blue, Purple};
```

ЗАПОМНИ!

C# предлагает несколько способов доступа к перечислению. Если вам нужно только одно отдельное значение, вы можете использовать имя цвета. Вывод, который вы получите, зависит от того, как вы обращаетесь к значению, как показано здесь:

```
// Вывод имени цвета.
Console.WriteLine(Colors.Blue);
// Вывод значения цвета.
Console.WriteLine((int)Colors.Blue);
```

Выполнив этот код, вы увидите в качестве выходных данных для первой строки Blue и для второй строки — 4. При создании перечисления значения перечислителей начинаются с 0 и последовательно увеличиваются. Поскольку Blue является пятым элементом, его значение равно 4.

В какой-то момент вам может понадобиться получить доступ ко всему списку перечисляемых значений. Для решения этой задачи можно использовать цикл foreach, например:

```
// Вывод всех элементов по именам
foreach (String Item in Enum.GetNames(typeof(Colors)))
    Console.WriteLine(Item);

// Вывод как имен, так и значений элементов.
foreach (Colors Item in Enum.GetValues(typeof(Colors)))
    Console.WriteLine("{0} = {1}", Item, (int)Item);
```

Однако вам может потребоваться только диапазон значений. В этом случае можно использовать цикл for, например:

```
// Вывод диапазона имен.
for (Colors Item = Colors.Orange; Item <= Colors.Blue; Item++)
    Console.WriteLine("{0} = {1}", Item, (int)Item);
```

В этом случае вы увидите только запрошенный диапазон предоставляемых перечислением Colors значений. Этот код дает следующий вывод:

```
Orange = 1
Yellow = 2
Green = 3
Blue = 4
```

Создание перечислений с инициализаторами

Использование значений по умолчанию, предоставляемых ключевым словом enum, в большинстве случаев вполне устраивает программиста, потому что его волнует не конкретное значение, а его удобочитаемая форма. Однако иногда действительно требуется назначить конкретные значения каждому из элементов перечисления. В этом случае вам нужен инициализатор. *Инициализатор* просто указывает конкретное значение, назначенное каждому элементу элемента:

```
enum Colors2
{
    Red    = 5,
    Orange = 10,
    Yellow = Orange + 5,
    Green  = 5 * 4,
    Blue   = 0x19,
    Purple = Orange | Green
}
```

Чтобы присвоить значение, просто добавьте знак равенства, а затем — числовое значение. Вы должны предоставить именно числовое значение, например вы не можете присвоить значение "Hello" одному из элементов.

ЗАПОМНИ!

Вы можете подумать, что последние четыре инициализатора выглядят странно. Но дело в том, что инициализатор может быть выражением, вычисление которого дает число. В первом случае вы добавляете 5 к значению Orange, чтобы инициализировать Yellow. Green представляет собой результат умножения. Blue использует шестнадцатеричный формат записи вместо десятичного. Наконец Purple является результатом применения логического ИЛИ к Orange и Green. К перечислениям, использующим инициализаторы, можно применять все те же методы, что и ранее:

```
foreach (Colors2 Item in Enum.GetValues(typeof(Colors2)))
    Console.WriteLine("{0} = {1}", Item, (int)Item);
```

Вот результат выполнения этих строк:

```
Red = 5
Orange = 10
Yellow = 15
Green = 20
Blue = 25
Purple = 30
```

Указание типа данных перечисления

Тип данных перечисления по умолчанию — `int`. Однако вы можете не захотеть использовать `int`; вам может понадобиться какое-то другое значение, такое как `long` или `short`. На самом деле вы можете использовать для создания перечисления типы `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong`. Тип, который вы выбираете, зависит от того, как вы планируете использовать перечисление и сколько значений вы планируете в нем хранить.

Чтобы определить тип данных `enum`, вы должны добавить двоеточие и имя типа после имени перечисления:

```
enum Colors3: byte {Red, Orange, Yellow, Green, Blue, Purple};
```

Перечисление `Colors3` предположительно имеет тип `byte`. Вы не узнаете этого наверняка, пока не выполните проверку этого факта. Следующий код показывает, как выполнить соответствующее тестирование:

```
foreach (Colors3 Item in Enum.GetValues(typeof(Colors3)))
    Console.WriteLine("{0} представляет собой {1} = {2}",
        Item, Item.GetTypeCode(), (int)Item);
```


ЗАПОМНИ!

Обратите внимание, что для получения типа, лежащего в основе перечисления, вы должны использовать метод `Item.GetTypeCode()`, а не `Item.GetType()`. Если вы используете `Item.GetType()`, то C# сообщит, что `Item` имеет тип `Colors3`. Вот вывод данного примера:

```
Red представляет собой Byte = 0
Orange представляет собой Byte = 1
Yellow представляет собой Byte = 2
Green представляет собой Byte = 3
Blue представляет собой Byte = 4
Purple представляет собой Byte = 5
```

Создание флагов-перечислений

Флаги предоставляют интересный способ работы с данными. Вы можете использовать их различными способами для выполнения таких задач, как определение параметров, не являющихся взаимоисключающими. Например, вы можете купить автомобиль с кондиционером, GPS, Bluetooth и рядом других функций. Каждая из этих функций является дополнением, но все они попадают в одну категорию дополнительных аксессуаров.


ЗАПОМНИ!

Работая с флагами, вы должны рассматривать их с точки зрения битов. Например, большинство людей считают, что `byte` может содержать значения до 255, но можно сказать иначе — что `byte` имеет

длину восемь битов. При работе с флагами необходим именно последний подход — что byte может содержать восемь отдельных битовых значений. Таким образом, значение 1 может указывать на то, что человек хочет кондиционер. Значение 2 может указывать на желание получить GPS. Точно так же значение 4 может указывать на необходимость Bluetooth — при использовании битовых позиций, показанных далее:

```
0000 0001  Кондиционер
0000 0010  GPS
0000 0100  Bluetooth
```

Зарезервировав битовые позиции и связав их с определенным выбором, вы можете начать битовые манипуляции, используя операторы и (&), или (|) и исключающее или (^). Например, значение 3, равное 00000011, скажет продавцу, что покупателю нужны кондиционер и GPS.

ЗАПОМНИ!

Наиболее распространенный способ работы со значениями битов — использование шестнадцатеричных значений, которые могут непосредственно представлять 16 различных значений, соответствующих четырем битовым позициям. Так, 0x11 в битовой записи представляет собой 00010001. Шестнадцатеричные значения находятся в диапазоне от 0 до F, где A = 10, B = 11, C = 12, D = 13, E = 14 и F = 15 в десятичной записи. Вот пример перечисляемого флага:

```
[Flags]
enum Colors4
{
    Red    = 0x01,
    Orange = 0x02,
    Yellow = 0x04,
    Green  = 0x08,
    Blue   = 0x10,
    Purple = 0x20
}
```

Обратите внимание на атрибут [Flags], находящийся непосредственно перед ключевым словом enum. Атрибут говорит компилятору C# о том, как реагировать на структуру особым образом. В данном случае вы говорите компилятору C#, что это не обычное перечисление; это перечисление определяет значения флагов.

Вы также должны были заметить, что отдельные элементы используют шестнадцатеричные инициализаторы (подробности см. в разделе "Создание перечислений с инициализаторами" выше в этой главе). C# не требует, чтобы вы использовали шестнадцатеричные инициализаторы, но это значительно

облегчает чтение вашего кода. Приведенный далее код показывает, как может работать флаг-перечисление:

```
// Создаем переменную с тремя вариантами цвета.
Colors4 myColors = Colors4.Red | Colors4.Green | Colors4.Purple;
// Выводим результат.
Console.WriteLine(myColors);
Console.WriteLine("0x{0:X2}", (int)myColors);
```

Этот код начинается с создания переменной `myColors`, которая содержит три выбора — `Colors4.Red`, `Colors4.Green` и `Colors4.Purple`. Чтобы создать аддитивный список выбора, значения комбинируются с использованием оператора `|`. Обычно `myColors` будет содержать значение `41`. Однако следующие две строки кода показывают влияние атрибута `[Flags]`:

```
Red, Green, Purple
0x29
```

Вывод показывает отдельные параметры при выводе `myColors`. Поскольку `myColors` представляет значения флагов, в примере также выводится значение `myColors`, равное `41`, в виде шестнадцатеричного значения `0x29`. Добавление форматной строки `X2` к аргументу формата приводит к выводу значения в шестнадцатеричном, а не десятичном виде, причем с двумя значащими цифрами. Аргумент формата и форматная строка разделяются двоеточием (`:`). Больше о типах форматов (включая форматные строки) вы можете узнать по адресу `https://docs.microsoft.com/dotnet/standard/base-types/formatting-types`.

Применение перечислений в конструкции `switch`

При работе с конструкцией `switch` причина принятия того или иного решения может оставаться неясной, если использовать числовое значение. Например, следующий код на самом деле мало что говорит вам о процессе принятия решений:

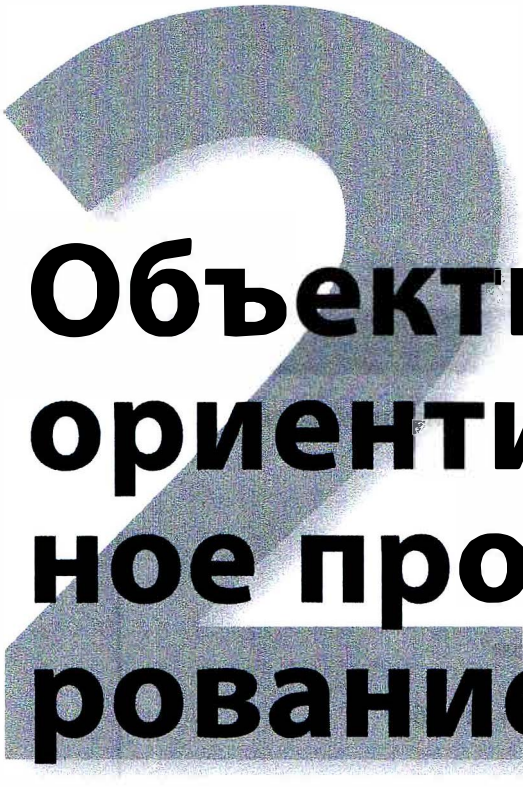
```
// Создание неясной конструкции switch.
int mySelection = 2;
switch (mySelection)
{
    case 0:
        Console.WriteLine("Выбран красный.");
        break;
    case 1:
        Console.WriteLine("Выбран оранжевый.");
```

```
        break;
    case 2:
        Console.WriteLine("Выбран желтый.");
        break;
    case 3:
        Console.WriteLine("Выбран зеленый.");
        break;
    case 4:
        Console.WriteLine("Выбран синий.");
        break;
    case 5:
        Console.WriteLine("Выбран фиолетовый.");
        break;
}
```

Этот код заставляет задуматься, почему `mySelection` присвоено значение `2` и о чем говорят эти инструкции вывода. Код работает, но почему и как — кажется тайной. Чтобы сделать код более удобочитаемым, можно использовать `switch` с перечислением:

```
// Создание более понятной конструкции switch.
Colors myColorSelection = Colors.Yellow;
switch (myColorSelection)
{
    case Colors.Red:
        Console.WriteLine("Выбран красный.");
        break;
    case Colors.Orange:
        Console.WriteLine("Выбран оранжевый.");
        break;
    case Colors.Yellow:
        Console.WriteLine("Выбран желтый.");
        break;
    case Colors.Green:
        Console.WriteLine("Выбран зеленый.");
        break;
    case Colors.Blue:
        Console.WriteLine("Выбран синий.");
        break;
    case Colors.Purple:
        Console.WriteLine("Выбран фиолетовый.");
        break;
}
```

Вывод в обоих случаях одинаковый: `"Выбран желтый."`. Однако во втором случае код куда более удобочитаем. Просто взглянув на код, вы узнаете, что `myColorSelection` содержит присвоенное ему значение цвета. Кроме того, использование члена `Colors` для каждой инструкции `case` делает выбор очевидным. Вы понимаете, почему код выбирает тот или иной определенный путь выполнения.



Объектно-ориентированное программирование на C#

В ЭТОЙ ЧАСТИ...

- Глава 11, "Что такое объектно-ориентированное программирование"
- Глава 12, "Немного о классах"
- Глава 13, "Методы"
- Глава 14, "Поговорим об этом"
- Глава 15, "Класс: каждый сам за себя"
- Глава 16, "Наследование"
- Глава 17, "Полиморфизм"
- Глава 18, "Интерфейсы"
- Глава 19, "Делегирование событий"
- Глава 20, "Пространства имен и библиотеки"
- Глава 21, "Именованные и необязательные параметры"
- Глава 22, "Структуры"

Глава 11

Что такое объектно-ориентированное программирование

В ЭТОЙ ГЛАВЕ...

- Основы объектно-ориентированного программирования
- Абстракция и классификация
- Важность объектно-ориентированного программирования

Объектная ориентированность является неотъемлемой частью современного программирования, поскольку помогает моделировать реальный мир с помощью кода, являющегося способом решения сложных задач программирования кодирования лучшим, чем применение методов процедурного программирования. В этой главе дается ответ на два основных вопроса: "Каковы концепции, лежащие в основе объектно-ориентированного программирования, и как они отличаются от процедурных концепций, описанных в части 1, "Основы программирования на C#", этой книги?" Мы начнем с рассмотрения абстракции, которую объектно-ориентированный подход предлагает для воспроизведения модели реального мира в коде. Далее в главе рассматриваются вопрос классификации и применение объектно-ориентированного подхода для более быстрого и простого (по сравнению с процедурными технологиями) создания приложений.

Объектно-ориентированная концепция № 1: абстракция

Вы садитесь в машину, намереваясь куда-то ехать. Вы запускаете двигатель, включаете передачу, а затем нажимаете акселератор, чтобы начать движение. Тормоз позволяет временно останавливать автомобиль, чтобы не сбивать пешеходов и не нарушать правила дорожного движения. Прибыв в пункт назначения, вы глушите мотор. Во всем этом сценарии есть вещи, которые вы *не* делаете:

- Вы не открываете моторный отсек, чтобы вручную регулировать поток топлива и воздуха и контролировать скорость двигателя.
- Вы не изменяете способ управления автомобилем таким образом, чтобы нажатие на акселератор останавливало автомобиль, а нажатие на тормоз заставляло его двигаться вперед.
- Вы не создаете совершенно новую систему сигналов о ваших намерениях другим водителям.

ЗАПОМНИ!

Это не просто пространные рассуждения. В повседневной жизни нас постоянно преследуют стрессы. Чтобы уменьшить их число, мы начинаем обращать внимание только на события определенного уровня детализации. В объектно-ориентированном программировании уровень детализации, на котором вы работаете, называется *уровнем абстракции*. И объяснить этот термин можно на примере *абстрагирования* от подробностей внутреннего устройства автомобиля.

К счастью, ученые-кибернетики — и тысячи фанатов программирования — открыли объектную ориентированность и ряд других концепций, снижающих уровень сложности, с которым должен работать программист. Использование абстракций делает программирование более простым и уменьшает количество возможных ошибок. Именно в этом направлении как минимум полстолетия движется прогресс в программировании — работа со все более сложными концепциями со все меньшим количеством ошибок.

Ведя машину, вы рассматриваете ее как черный ящик. (По дороге на рынок или в театр вы не можете позволить себе беспокоиться о внутренностях автомобиля и одновременно избегать столкновений с этими надоедливыми пешеходами.) Пока вы пользуетесь автомобилем только с помощью его интерфейса (различных элементов управления), что бы вы ни делали, это не приведет к тому, что автомобиль войдет в противоречивое состояние и взорвется.

Процедурные поездки

Предположим, вы хотите создать процедуру использования автомобиля для небольшой поездки. Вы можете включить в процедуру следующие элементы:

1. завести машину;
2. выехать на полосу движения;
3. ехать в нужное место,
4. избегая всех препятствий на пути и
5. следуя правилам дорожного движения;
6. по прибытии припарковать машину;
7. выключить двигатель.

Это описание простое и полное. Но с помощью подобного алгоритма "функциональный" программист не сможет написать программу для поездки. Процедурные программисты живут в мире, лишенном таких объектов, как автомобили, правила дорожного движения и пешеходы. Они склонны беспокоиться о блок-схемах с их бесчисленными процедурными путями. В процедурном решении задачи поездки программисту необходимо описать фактические взаимодействия между компонентами, например как нажатие на акселератор увеличивает поток топлива. И очень быстро весь процесс поездки превращается в кошмар маленьких взаимодействий, которые не имеют никакого отношения к вождению.

В мире процедурного программирования вы не можете отвлечься от деталей и думать в терминах уровней абстракции. У вас нет объектов и абстракций, за которыми скрыта вся сложность.

Объектно-ориентированные поездки

В объектно-ориентированном подходе к поездке вы сначала определяете типы объектов в задаче: автомобиль, акселератор, тормоз, указатели поворота, пешеходы и т.д. Затем вы приступаете к моделированию этих объектов в программном обеспечении, не обращая внимания на детали их использования в конечной программе. Например, вы можете смоделировать акселератор как объект, изолированный от других объектов, а затем объединить его с тормозом, указателем поворота и другими компонентами, чтобы заставить их взаимодействовать. (И вы можете решить, что некоторые из этих объектов не обязательно должны быть объектами в вашей программе, например пешеходы.)

Выполняя все это, вы работаете (и думаете) на уровне базовых объектов. Вам нужно думать о создании полезного автомобиля, а не о процессе нажатия на педаль акселератора (пока что). В конце концов, проектировщики

автомобилей не думают о конкретной задаче сделать левый поворот — они решают проблему проектирования и постройки полезного (а главное, хорошо продаваемого) автомобиля.

Успешно закодировав и протестировав нужные вам объекты, вы можете перейти к следующему уровню абстракции и начать думать на уровне поездки, а не на уровне производства автомобиля. С этого момента вы можете начать создавать автомобиль своей мечты, чтобы совершить поездку в ваше любимое место.

Объектно-ориентированная концепция № 2: классификация

Критическим для концепции абстракции является понятие классификации. Обсуждая автомобиль, легко увидеть, что это просто колесное транспортное средство. Однако оно имеет определенные характеристики. Например, автомобиль с четырьмя колесами будет работать, в отличие от автомобиля только с двумя колесами. С другой стороны, мотоцикл тоже является разновидностью колесного транспортного средства, но он отлично работает, используя только два колеса. *Классификация* помогает определить типы объектов для моделирования в программном обеспечении.

ЗАПОМНИ!

В объектно-ориентированном программировании автомобиль является *экземпляром* класса `car`. Класс `car` является *подклассом* класса `wheeledVehicle` (колесное транспортное средство), который является подклассом класса `vehicle` (транспортное средство). Аналогично мотоцикл является экземпляром класса `motorcycle`, который также является подклассом `wheeledVehicle`, который является подклассом `vehicle`. Однако лодка будет экземпляром класса `boat`, который является подклассом `floatingVehicle` (плавающее транспортное средство), который является подклассом `vehicle`. Таким образом, автомобили, мотоциклы и лодки — все они являются транспортными средствами, но это особые разновидности транспортных средств.

Люди склонны заниматься классификацией. Все вокруг увешано ярлыками. Мы делаем все для того, чтобы уменьшить количество вещей, которые надо запомнить. Вспомните, например, как вы впервые увидели "Пежо" или "Рено". Возможно, в рекламе и говорилось, что это супераавтомобиль, но мы-то с вами знаем, что это не так. Это ведь просто машина. Она имеет все свойства, которыми обладает автомобиль. У нее есть руль, колеса, сиденья, мотор, тормоза и т.д. И можно поспорить, что многие водили бы такую штуку без всяких инструкций.

Но не будем тратить место в книге на описание того, чем этот автомобиль похож на другие. Следует знать лишь то, что это "машина, которая...", и то, чем она отличается от других машин (например, ценой).

Зачем нужна классификация

Зачем вообще нужны вся эта классификация и это объектно-ориентированное программирование? Ведь оно влечет за собой массу трудностей. Тем более что уже есть готовый механизм функций. Зачем же что-то менять?

Проектирование и сборка автомобиля специально для местных поездок (а не для длительных поездок) может показаться проще, чем создание отдельного, более универсального объекта колесного транспортного средства. Предположим, что вы хотите построить автомобиль, у которого есть только один элемент управления движением, а не отдельные элементы управления акселератором и тормозом, которые в значительной степени необходимы при движении на большие расстояния. Вы можете даже не монтировать эти элементы управления на пол; они могут иметь вид рычага рядом с рукой водителя (аналогично тем, которые используются садовыми тракторами и косилками). К сожалению, такое транспортное средство, хотя и более простое, будет одновременно и менее функциональным. Процедурный подход имеет следующие проблемы.

- **Слишком сложен.** Нежелательно, чтобы детали при построении автомобиля перемешивались с деталями путешествия на нем. Но поскольку при данном подходе нельзя создавать объекты и упрощать написание программы, работая с каждым из них в отдельности, приходится держать в голове все сложности задачи одновременно.
- **Не гибок.** Когда-нибудь вам, возможно, придется заменить автомобиль другим типом колесного транспортного средства или, возможно, летающим транспортным средством. Это должно легко получиться, если два транспортных средства имеют один и тот же интерфейс (или если летающее транспортное средство является истинным *надмножеством*, имеющим дополнительные элементы управления по сравнению с колесным транспортным средством). Не будучи четко описанным и отдельно разработанным, один тип объекта не может быть полностью удален и заменен другим.
- **Невозможность повторного использования.** Автомобили используются для многих различных типов поездок. Вы же не хотите создавать новую машину каждый раз, когда нужно поехать в новое место? Решив проблему один раз, вы хотите иметь возможность повторно использовать решение и в других местах программы. Если вам повезет, вы сможете использовать его и в будущих программах.

Объектно-ориентированная концепция № 3: удобные интерфейсы

Объект должен быть способен спроектировать внешний интерфейс максимально простым при полной достаточности для корректного функционирования. Если интерфейс устройства будет недостаточен, все закончится стучанием кулаком или чем-то более тяжелым по верхней панели такого устройства или просто разборкой для того, чтобы добраться до его внутренностей. Знание того, что устройство имеет возможности, недостижимые через интерфейс, разочаровывает. С другой стороны, если интерфейс устройства слишком сложный, никто не купит такое устройство, или по крайней мере никто не будет использовать все его возможности.

Люди постоянно жалуются на сложность DVD-плееров (впрочем, с переходом на управление с помощью экрана количество жалоб несколько уменьшилось). В этих устройствах слишком много кнопок с различными функциями. Зачастую одна и та же кнопка выполняет разные функции в зависимости от того, в каком именно состоянии находится в этот момент плеер. Кроме того, похоже, невозможно найти два DVD-плеера различных марок с одинаковыми интерфейсами.

Теперь рассмотрим ситуацию с автомобилями. Вряд ли можно сказать (и доказать), что автомобиль проще плеера. Однако, похоже, люди не испытывают таких трудностей с вождением, как с управлением плеером.

В каждом автомобиле есть примерно одни и те же элементы управления и примерно в одних и тех же местах. Если же управление отличается… Вот вам реальная история из моей жизни. У моей сестры был французский автомобиль, в котором управление фарами оказалось там, где во всех "нормальных" автомобилях находится управление сигналами поворота. Отличие вроде бы небольшое, но я так и не научился поворачивать ночью на этом автомобиле влево, не выключив при этом фары…

Кроме того, при хорошо продуманном дизайне автомобиля один и тот же элемент управления никогда не будет использоваться для выполнения более одной операции в зависимости от состояния автомобиля. Конечно, есть и исключение из этого правила: некоторые кнопки на большинстве устройств круиз-контроля перегружены множеством функций.

Объектно-ориентированная концепция № 3: управление доступом

Некоторые устройства допускают несколько комбинаций управления доступом, например микроволновая печь. Микроволновая печь должна быть сконструирована таким образом, чтобы никакие комбинации нажатий клавиш, которые вы можете ввести на передней клавиатуре, не могли повредить ее. Конечно, некоторые комбинации ничего не делают. Однако никакая последовательность нажатий клавиш не должна приводить к следующему.

- **К поломке устройства.** Какие бы рукоятки ни крутил ваш ребенок и какие бы кнопки ни нажимал, микроволновая печь не должна от этого сломаться. После того как вы вернете все элементы управления в корректное состояние, она должна нормально работать, если, понятно, в приступе злости вы не швырнете ее о стену.
- **К пожару или прочей порче имущества или нанесению вреда здоровью потребителя.** Мы живем в сутяжном мире, и если бы что-то похожее могло произойти, компании пришлось бы продать все вплоть до автомобиля ее президента, чтобы рассчитаться с подающими на нее в суд истцами и адвокатами.

Однако, чтобы эти два правила выполнялись, необходимо принять на себя определенную ответственность. Вы ни в коем случае не должны вносить изменения в устройство, в частности отключать блокировки.

Почти все кухонное оборудование любой степени сложности, включая микроволновые печи, имеет пломбы, препятствующие проникновению пользователя внутрь. Если такая пломба повреждена, значит, крышка устройства была снята, и вся ответственность с производителя тем самым снимается. Если вы каким-либо образом изменили внутреннее устройство печи, вы сами несете ответственность за все последующие неприятности, которые могут произойти.

Аналогично класс должен иметь возможность контролировать доступ к своим членам-данным. Никакая последовательность вызовов членов класса не должна приводить программу к аварийному завершению, однако класс не в состоянии гарантировать это, если внешние объекты имеют доступ к внутреннему состоянию класса. Класс должен иметь возможность прятать критические члены-данные и делать их недоступными для внешнего мира.

Поддержка объектно-ориентированных концепций в C#

Итак, как же C# реализует объектно-ориентированное программирование? Впрочем, это не совсем корректный вопрос. C# является объектно-ориентированным языком программирования, но не реализует его — это делает программист. Как и на любом другом языке, вы можете написать на C# программу, не являющуюся объектно-ориентированной (например, вставив весь код Word в функцию `Main()`). Иногда нужно писать и такие программы, но все же главное назначение C# — создание объектно-ориентированных программ. Язык C# предоставляет программисту следующие необходимые для написания объектно-ориентированных программ возможности.

- **Управляемый доступ.** C# управляет обращением к членам класса. Ключевые слова C# позволяют объявить одни члены открытыми (`public`) для всех, а другие — защищенными (`protected`) или закрытыми (`private`). Подробнее эти вопросы рассматриваются в главе 15, "Класс: каждый сам за себя".
- **Специализация.** C# поддерживает специализацию посредством механизма, известного как *наследование классов*. Один класс при этом наследует члены другого класса. Например, вы можете создать класс `Car` как частный случай класса `Vehicle`. Подробнее эти вопросы рассматриваются в главе 16, "Наследование".
- **Полиморфизм.** Эта возможность позволяет объекту выполнить операцию так, как это требуется для его корректного функционирования. Например, класс `Rocket`, унаследованный от `Vehicle`, может реализовать операцию `Start` совершенно иначе, чем `Car`, унаследованный от того же `Vehicle`. Вопросы полиморфизма рассматриваются в главе 17, "Полиморфизм".
- **Косвенность.** Объекты часто используют функциональность других объектов — путем вызова их открытых методов. Но классы могут "слишком много знать" о классах, которые они используют. В таком случае говорят о том, что классы "слишком сильно связаны", что делает использующий класс чересчур зависящим от используемого. Такая конструкция очень хрупкая и может легко сломаться при внесении в нее небольших изменений. Но внесение изменений при программировании совершенно неизбежно, так что лучше всего найти более непрямые, *косвенные* (indirect), способы соединения двух классов. Именно здесь в игру вступают *интерфейсы* C# (о них вы узнаете из главы 18, "Интерфейсы").

Глава 12

Немного о классах

В ЭТОЙ ГЛАВЕ...

- Введение в классы C#
- Присваивание и использование ссылок на объекты
- Классы, содержащие классы
- Статические члены и члены экземпляров

Вы можете свободно объявлять и использовать все встроенные типы данных — такие, как `int`, `double` или `bool` — для хранения информации, необходимой вашей программе. Для ряда программ таких простых переменных вполне достаточно, но большинству программ требуется средство для объединения связанных данных в аккуратные пакеты.

Как уже говорилось в части I, "Основы программирования на C#", C# предоставляет возможность использовать массивы и коллекции для группирования в единую структуру *однотипных* переменных, таких как `string` или `int`. Взяв в качестве примера гипотетический колледж, можно разместить его студентов в массиве. Но как такая программа может представить студента? Ведь это существенно большее понятие, чем просто имя, — как программа должна представить такой сложный объект, каковым является студент?

Некоторым программам необходимо собрать вместе данные, связанные логически, но имеющие разные типы. Например, приложение, работающее со списками студентов, должно хранить разнотипную информацию о них — имя, год рождения, успеваемость и т.п. Логически рассуждая, имя студента может иметь тип `string`, год рождения — `int` или `short`, средний балл — `double`.

Такой программе необходима возможность объединить эти разнотипные переменные в единую структуру под именем `Student`. К счастью, в C# имеется структура, известная как *класс*, которая предназначена для облегчения группирования таких разнотипных переменных.

Определение класса и объекта

Класс представляет собой объединение разнотипных данных и функций, логически организованных в единое целое. C# предоставляет полную свободу при создании классов, но хорошо спроектированные классы призваны представлять *концепции*.

Программирование моделирует объекты реального мира с помощью структур, которые представляют концепции и объекты реального мира, такие как банковские счета, игры, покупатели, документы или товары. Аналитик, скорее всего, скажет, что "класс отображает концепцию из предметной области задачи в программу". Предположим, например, что ваша задача — построить имитатор дорожного движения, который должен смоделировать улицы, перекрестки, шоссе и т.п.

Любое описание такой задачи должно включать термин *транспортное средство*. Транспортные средства обладают определенной максимальной скоростью движения и имеют вес; некоторые из них оснащены прицепами. Кроме того, транспортное средство может стоять или передвигаться. Значит, в качестве концепции *транспортное средство* представляет собой часть предметной области.

Таким образом, хороший симулятор дорожного движения должен включать класс `Vehicle`, описывающий существенные для моделирования свойства транспортного средства. Такой класс `Vehicle` в C# будет иметь свойства наподобие `topSpeed`, `weight` и `isClunker`.

Поскольку классы — центральная концепция в программировании на C#, они будут гораздо детальнее рассмотрены в остальных главах части 2, "Объектно-ориентированное программирование на C#"; здесь же описаны только азы.

Определение класса

Пример класса `Vehicle` может выглядеть следующим образом:

```
public class Vehicle
{
    public string model;        // Название модели
    public string manufacturer; // Производитель
    public int numOfDoors;      // Количество дверей
    public int numOfWheels;     // Количество колес
}
```

Определение класса начинается словами `public class`, за которыми следует имя класса (в данном случае — `Vehicle`). Как и все имена в C#, имена классов чувствительны к регистру. C# не имеет никаких правил для именования классов, но неофициальная традиция гласит, что имена классов начинаются с прописной буквы.

За именем класса следует пара фигурных скобок, внутри которых может содержаться несколько *членов* (либо ни одного). Члены класса представляют собой переменные, образующие часть класса. В данном примере класс `Vehicle` начинается с члена `model` с типом `string`, который содержит название модели транспортного средства. Второй член в этом примере — `string manufacturer`, а последние два члена (оба типа `int`) содержат количество дверей и колес в транспортном средстве.


СОВЕТ

Как и в случае обычных переменных, делайте имена членов максимально информативными. Хорошее имя переменной говорит о ней и ее предназначении все. Добавление к именам членов комментариев, как показано в данном примере, может облегчить понимание назначения членов и правил их применения.

Модификатор `public` перед именем класса делает класс доступным для всей программы. Аналогично модификаторы `public` перед именами членов также делают их доступными для всей программы. Возможны и другие модификаторы, но более подробно о доступности и о том, как скрывать некоторые члены, речь пойдет в главе 15, "Класс: каждый сам за себя".

Определение класса должно описывать свойства объектов решаемой задачи. Сделать это прямо сейчас вам будет немного сложно, поскольку вы не знаете, в чем именно состоит задача, но все станет понятнее при работе над конкретной программой.

Что такое объект

Создать проект автомобиля — не то же самое, что и создать сам автомобиль. Кто-то должен отрезать лист металла, взять горсть болтов и соединить это все вместе, чтобы можно было куда-то поехать. Объект класса объявляется аналогично встроенным объектам C# наподобие `int`, но не идентично им.


ЗАПОМНИ!

Термин *объект* используется для обозначения "вещей". Да, это не слишком полезное определение, так что приведем несколько примеров. Переменная типа `int` является объектом `int`. Автомобиль является объектом `Vehicle`. Вот фрагмент кода, создающий автомобиль, который является объектом `Vehicle`:

```
Vehicle myCar;
myCar = new Vehicle();
```

В первой строке объявлена переменная `myCar` типа `Vehicle`, так же, как вы можете объявить переменную `somethingOrOther` класса `int` (да, класс является типом, и все объекты C# определяются как классы). Команда `new Vehicle()` создает конкретный объект типа `Vehicle` и сохраняет его местоположение в памяти в переменной `myCar`. Оператор `new` ("новый") не имеет ничего общего с возрастом автомобиля — он просто выделяет новый блок памяти, в котором ваша программа может хранить свойства автомобиля `myCar`.


ЗАПОМНИ!

В терминах C# `myCar` — это объект класса `Vehicle`. Можно также сказать, что `myCar` — экземпляр класса `Vehicle`. В данном контексте *экземпляр* (instance) означает "пример" или "один из". Можно использовать этот термин и как глагол и говорить об *инстанцировании* `Vehicle` (это именно то, что делает оператор `new`). Сравните объявление `myCar` с объявлением переменной `num` типа `int`:

```
int num;
num = 1;
```

В первой строке объявлена переменная `num` типа `int`, а во второй созданной переменной присваивается значение типа `int`, которое вносится в память по месту расположения переменной `num`.


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Переменная встроенного типа `num` и объект `myCar` хранятся в памяти по-разному. Первая использует *стек*, а вторая — динамическую память, *кучу*. Переменная `num` действительно содержит `1`, а не местоположение в памяти. Выражение `new Vehicle` выделяет необходимую память в куче. Переменная `myCar` содержит ссылку на память, а не фактические значения для описания `Vehicle`.

Доступ к членам объекта

Каждый объект класса `Vehicle` имеет собственный набор членов. Приведенное далее выражение сохраняет число `1` в члене `numberOfDoors` объекта, на который ссылается `myCar`:

```
myCar.numberOfDoors = 1;
```


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Каждая операция C# должна давать как значение, так и тип. Объект `myCar` является объектом типа `Vehicle`. Переменная `Vehicle.numberOfDoors` имеет тип `int` (вернитесь к определению класса `Vehicle`). Константа `1` также имеет тип `int`, так что тип константы справа от оператора присваивания и переменной слева соответствуют друг другу. Аналогично в следующем фрагменте кода

сохраняются ссылки на строки string, описывающие модель и производителя myCar:

```
myCar.manufacturer = "BMW";
myCar.model = "Isetta";
```

(Isetta — небольшой автомобиль, который производился в 1950-х годах и имел одну дверь впереди.)

Пример объектно-основанной программы

Программа VehicleDataOnly очень проста и делает следующее:

- определяет класс Vehicle;
- создает объект myCar;
- указывает свойства myCar;
- получает значения от объекта и выводит их на экран.

Вот код программы VehicleDataOnly.

```
// VehicleDataOnly
// Создает объект Vehicle, заполняет его члены информацией,
// вводимой с клавиатуры, и выводит ее на экран
using System;

namespace VehicleDataOnly
{
    public class Vehicle
    {
        public string model;          // Модель
        public string manufacturer;   // Производитель
        public int numOfDoors;        // Количество дверей
        public int numOfWheels;       // Количество колес
    }

    public class Program
    {
        // Начало программы
        static void Main(string[] args)
        {
            // Приглашение пользователю
            Console.WriteLine("Введите информацию о машине");
            // Создание экземпляра Vehicle
            Vehicle myCar = new Vehicle();
            // Ввод информации для членов класса
            Console.Write("Модель        = ");
            string s = Console.ReadLine();
            myCar.model = s;
```

```
            // Можно присваивать значения непосредственно
            Console.Write("Производитель = ");
            myCar.manufacturer = Console.ReadLine();
            // Остальные данные имеют тип int
            Console.Write("Количество дверей = ");
            s  = Console.ReadLine();
            myCar.numOfDoors = Convert.ToInt32(s);
            Console.Write("Количество колес  = ");
            s  = Console.ReadLine();
            myCar.numOfWheels = Convert.ToInt32(s);
            // Вывод полученной информации
            Console.WriteLine("\nВаша машина: ");
            Console.WriteLine(myCar.manufacturer  + " " +
                              myCar.model);
            Console.WriteLine("с " + myCar.numOfDoors +
                              " дверями, "
                              +  "на " + myCar.numOfWheels
                              +  " колесах");
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

ЗАПОМНИ!

Листинг программы начинается с определения класса `Vehicle`. Определение класса может находиться как до, так и после класса `Program` (это не имеет значения). Однако вам нужно выбрать один стиль и постоянно следовать ему.

Программа создает объект `myCar` класса `Vehicle`, а затем заполняет все его поля информацией, вводимой пользователем с клавиатуры. Проверка корректности входных данных не выполняется. Затем программа выводит введенные данные на экран в несколько ином формате. Вывод этой программы выглядит следующим образом:

```
Введите информацию о машине
Модель        = Metropolitan
Производитель = Nash
Количество дверей = 2
Количество колес  = 4

Ваша машина:
Nash Metropolitan
с 2 дверями на 4 колесах
Нажмите <Enter> для завершения программы...
```

СОВЕТ

Вызов `Write()`, в отличие от `WriteLine()`, оставляет курсор сразу за введенной строкой. Это приводит к тому, что ввод пользователя находится в той же строке, где и приглашение. Кроме того, добавление символа новой строки `'\n'` создает пустую строку без необходимости вызывать `WriteLine()`.

Различие между объектами

Заводы в Детройте в состоянии выпускать множество автомобилей, отслеживать каждую выпущенную машину и при этом не путать их. Аналогично программа может создать несколько объектов одного и того же класса, как показано в следующем фрагменте:

```
Vehicle car1 = new Vehicle();
car1.manufacturer = "Studebaker";
car1.model = "Avanti";

// Следующий код никак не влияет на car1
Vehicle car2 = new Vehicle();
car2.manufacturer = "Hudson";
car2.model = "Hornet";
```

Создание объекта `car2` и присваивание ему имени производителя `Hudson` никак не влияют на объект `car1` с именем производителя `Studebaker`. Это связано с тем, что `car1` и `car2` находятся в разных местах памяти. Возможность различать объекты одного класса очень важна в работе с классами. Объект может быть создан, с ним могут быть выполнены различные действия и он всегда выступает как единый объект, отличный от других подобных ему объектов.

Работа со ссылками

Оператор "точка" и оператор присваивания — это операторы, определенные для ссылочных типов. Рассмотрим следующий фрагмент исходного текста:

```
// Создание нулевой ссылки
Vehicle yourCar;

// Присваивание значения ссылке
yourCar = new Vehicle();

// Использование точки для обращения к члену
yourCar.manufacturer = "Rambler";

// Создание новой ссылки, которая указывает на тот же объект
Vehicle yourSpousalCar = yourCar;
```

В первой строке создается объект `yourCar`, причем без присваивания ему значения. Такая неинициализированная ссылка называется *нулевым объектом* (null object). Любые попытки использовать неинициализированную ссылку приводят к немедленной генерации ошибки, которая прекращает выполнение программы.

Компилятор C# может перехватить большинство попыток использования неинициализированной ссылки и сгенерировать предупреждение в процессе компиляции программы. Если вам каким-то образом удалось провести компьютер, то обращение к неинициализированной ссылке при выполнении программы приведет к ее аварийному останову.

Второе выражение создает новый объект `Vehicle` и присваивает его ссылочной переменной `yourCar`. И последняя строка кода присваивает ссылке `yourSpousalCar` ссылку `yourCar`. Как показано на рис. 12.1, это приводит к тому, что `yourSpousalCar` ссылается на тот же объект, что и `yourCar`.

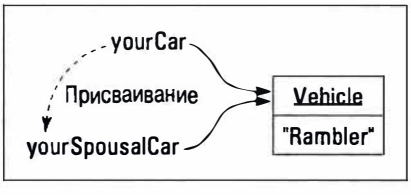


Рис. 12.1. Две ссылки на один и тот же объект

Эффект от следующих двух вызовов одинаков:

```
// Создание вашей машины
Vehicle yourCar = new Vehicle();
yourCar.model = "Kaiser";

// Эта машина принадлежит и вашей жене
Vehicle yourSpousalCar = yourCar;

// Изменяя одну машину, вы изменяете и другую
yourSpousalCar.model = "Henry J";
Console.WriteLine("Ваша машина - " + yourCar.model);
```

Выполнение данной программы приводит к выводу на экран названия модели `Henry J`, а не `Kaiser`. Обратите внимание на то, что `yourSpousalCar` не указывает на `yourCar` — вместо этого и `yourSpousalCar`, и `yourCar` указывают на один и тот же объект (одно и то же место в памяти). Кроме того, ссылка `yourSpousalCar` будет корректной, даже если окажется "потерянной" (например, при выходе за пределы области видимости), как показано в следующем фрагменте:

```
// Создание вашей машины
Vehicle yourCar = new Vehicle();
yourCar.model = "Kaiser";

// Эта машина принадлежит и вашей жене
Vehicle yourSpousalCar = yourCar;

// Когда она забирает себе вашу машину...
yourCar = null;      // yourCar теперь ссылается на "нулевой
                     // объект"

// ...yourSpousalCar ссылается на все ту же машину
Console.WriteLine("Ваша машина - " + yourSpousalCar.model);
```

В результате выполнения этого фрагмента исходного текста на экран выводится сообщение "Ваша машина - Kaiser", несмотря на то что ссылка yourCar стала недействительной. Объект перестал быть *достижимым* по ссылке yourCar. Но он не будет полностью недостижимым, пока не будут "потеряны" или обнулены обе ссылки — и yourCar, и yourSpousalCar.

После этого — вернее будет сказать, в некоторый непредсказуемый момент после этого — *сборщик мусора* (garbage collector) C# вернет память, использованную ранее для объекта, все ссылки на который утрачены.

СОВЕТ

Возможность сделать одну *объектную переменную* (переменную ссылочного типа, такого как Vehicle или Student, в отличие от простого типа, такого как int или double) указывающей на другой объект — как это было сделано выше — делает работу со ссылочными объектами в массивах и коллекциях и их хранение очень эффективными. Каждый элемент массива хранит ссылку на объект, и когда вы обмениваете элементы в массиве, то перемещаются только ссылки, но не сами объекты. Ссылки занимают фиксированное количество памяти, в отличие от объектов, на которые указывают.

Классы, содержащие классы

Члены класса могут, в свою очередь, быть ссылками на другие классы. Например, транспортное средство имеет двигатель, свойствами которого являются, в частности, мощность и рабочий объем. Можно поместить эти параметры непосредственно в класс Vehicle следующим образом:

```
public class Vehicle
{
    public string model;        // Модель
    public string manufacturer; // Производитель
    public int numOfDoors;      // Количество дверей
    public int numOfWheels;     // Количество колес
```

```
    // Новые члены:
    public int power;           // Мощность двигателя
    public double displacement; // Рабочий объем
}
```

Однако мощность и рабочий объем двигателя не являются свойствами автомобиля, так как, например, джип моего сына может поставляться с одним из двух двигателей с совершенно разными мощностями. Двигатель является самодостаточной концепцией и может быть описан отдельным классом:

```
public class Motor
{
    public int power;           // Мощность
    public double displacement; // Рабочий объем
}
```

Вы можете внести этот класс в класс `Vehicle` следующим образом:

```
public class Vehicle
{
    public string model;        // Модель
    public string manufacturer; // Производитель
    public int numOfDoors;      // Количество дверей
    public int numOfWheels;     // Количество колес
    public Motor motor;
}
```

Соответственно, создание `sonsCar` теперь выглядит так:

```
// Сначала создаем двигатель
Motor largerMotor = new Motor();
largerMotor.power = 230;
largerMotor.displacement = 4.0;

// Теперь создаем автомобиль
Vehicle sonsCar = new Vehicle();
sonsCar.model = "Cherokee Sport";
sonsCar.sManfacturer = "Jeep";
sonsCar.numOfDoors = 2;
sonsCar.numOfWheels = 4;

// Присоединяем двигатель к автомобилю
sonsCar.motor = largerMotor;
```

Доступ к рабочему объему двигателя из `Vehicle` можно получить в два этапа, как показано в приведенном далее фрагменте:

```
Motor m = sonsCar.motor;
Console.WriteLine("Рабочий объем равен " + m.displacement);
```

Однако можно получить эту величину и непосредственно:

```
Console.WriteLine("Рабочий объем равен " +
                  sonsCar.motor.displacement);
```

В любом случае доступ к значению `displacement` осуществляется через класс `Motor`.

Статические члены класса

Большинство членов-данных описывают отдельные объекты. Рассмотрим следующий класс `Car`:

```
public class Car
{
    public string licensePlate; // Номерной знак автомобиля
}
```

Номерной знак является *свойством объекта*, описывающим каждый автомобиль и уникальным для каждого автомобиля. Присваивание номерного знака одному автомобилю не меняет номерной знак другого:

```
Car spouseCar = new Car();
spouseCar.licensePlate = "XYZ123";

Car yourCar = new Car();
yourCar.licensePlate = "ABC789";
```

Однако имеются и такие свойства, которые присущи всем автомобилям. Например, количество выпущенных автомобилей является свойством класса `Car`, но не отдельного объекта. *Свойство класса* помечается специальным ключевым словом `static`, как показано в следующем фрагменте исходного текста:

```
public class Car
{
    public static int numberOfCars; // Выпущено автомобилей
    public string licensePlate;     // Номерной знак автомобиля
}
```

ЗАПОМНИ!

Обращение к статическим членам выполняется не посредством объекта, а через сам класс, как показано в следующем фрагменте исходного текста:

```
// Создание нового объекта класса Car
Car newCar = new Car();
newCar.licensePlate = "ABC123";

// Увеличиваем количество автомобилей на 1
Car.numberOfCars++;
```

Обращение к члену объекта `newCar.licensePlate` выполняется посредством объекта `newCar`, в то время как обращение к (статическому) члену `Car.numberOfCars` осуществляется с помощью имени класса. Все объекты типа `Car` совместно используют один и тот же член `numberOfCars`, так что каждый автомобиль содержит то же самое значение этого члена, что и все прочие автомобили.

ЗАПОМНИ!

Члены класса являются статическими членами. Нестатические члены свои для каждого "экземпляра" (каждого отдельного объекта) и называются *членами экземпляра*. Выделенные курсивом термины являются распространенным названием этих видов членов.

Определение константных членов-данных и членов-данных только для чтения

Специальным видом статических членов является член, представляющий собой константу. Значение такой константной переменной должно быть указано в объявлении и не может изменяться нигде в программе, как показано в следующем фрагменте исходного текста:

```
class Program
{
  // Число дней в году (включая високосный год)
  public const int daysInYear = 366; // Должен иметься
                                     // Инициализатор
  public static void Main(string[] args)
  {
    // Это массив (о нем будет рассказано немного позже)
    int[] nMaxTemperatures = new int[daysInYear];
    for(int index = 0; index < daysInYear; index++)
    {
      // Вычисление средней температуры для каждого дня года
    }
  }
}
```

Константу `daysInYear` можно использовать везде в вашей программе вместо числа 366. Константные переменные очень полезны, так как позволяют заменить "магические числа" (в данном случае — 366) описательным именем `daysInYear`, что повышает удобочитаемость программы и облегчает ее сопровождение. C# предоставляет и другой способ объявления констант — можно предварить объявление переменной модификатором `readonly`:

```
public readonly int daysInYear = 366; // Может быть статическим членом
```

Как и при применении модификатора `const`, значение такого члена (после того, как вы присвоите ему инициализирующее значение) не может быть изменено нигде в программе. Хотя причины этого совета носят слишком технический характер, чтобы описывать их в настоящей книге, при объявлении констант предпочтительно использовать модификатор `readonly`.

Модификатор `const` можно использовать с данными-членами класса и внутри методов класса; однако модификатор `readonly` в методах недопустим (о методах подробнее будет рассказано в главе 13, "Методы").

Для констант имеется собственное соглашение об именовании. Многие программисты вместо их именования так же, как и переменных (как в примере с `daysInYear`), предпочитают использовать прописные буквы с разделением слов символами подчеркивания — `DAYS_IN_YEAR`. Такое соглашение позволяет отделить константы от обычных изменяемых переменных.

Глава 13

Методы

В ЭТОЙ ГЛАВЕ...

- Определение метода
- Передача аргументов методу
- Получение результатов из метода
- Метод `WriteLine()`

Программисты должны иметь возможность разбивать большие программы на части, с которыми легко работать. Например, программы, содержащиеся в предыдущих главах этой мини-книги, достигают предела количества информации, которую человек может усвоить за один раз.

ЗАПОМНИ!

C# позволяет разделить код класса на фрагменты, известные как *методы*. Метод эквивалентен функции, процедуре или подпрограмме в других языках. Разница в том, что метод всегда является частью класса. Правильно спроектированные и реализованные методы могут значительно упростить работу по написанию сложных программ.

Определение и использование метода

Рассмотрим следующий пример:

```
class Example
{
    public int anInt;                // Не статический член
    public static int staticInt      // Статический член
    public void MemberMethod()       // Не статический метод
    {
        Console.WriteLine("Это метод экземпляра");
    }
    public static void ClassMethod() // Статический метод
    {
        Console.WriteLine("Это метод класса");
    }
}
```

Элемент `anInt` является членом-данными, с которыми вы познакомились в части 1, "Основы программирования на C#". Однако элемент `MemberMethod()` для вас нов. Он известен как *метод экземпляра* или *функция-член*, представляющая собой набор кода C#, который может быть выполнен с помощью ссылки на имя этого метода. Честно говоря, такое определение смущает даже меня, так что лучше рассмотреть, что такое метод, на примерах. (`Main()` и `WriteLine()` используются почти в каждом примере книги и являются методами.)

Примечание. Различие между статическими и нестатическими методами крайне важно. Частично данная тема будет раскрыта в настоящей главе, но более подробно об этом речь пойдет в главе 14, "Поговорим об этом", в которой будут более детально рассмотрены нестатические методы.

ЗАПОМНИ!

Для вызова нестатического метода необходим экземпляр класса. Для вызова статического метода требуется имя класса, а не экземпляр. В следующем фрагменте присваиваются значения члену объекта `anInt` и члену класса (статическому члену) `staticInt`:

```
Example example    = new Example(); // Создание объекта
example.anInt      = 1;            // Инициализация члена с
                                   // использованием объекта
Example.staticInt = 2;             // Инициализация члена с
                                   // использованием класса
```

Практически аналогично в приведенном далее фрагменте происходит обращение (путем вызова) к методам `InstanceMethod()` и `ClassMethod()`:

```
Example example = new Example(); // Создание экземпляра
example.InstanceMethod();        // Вызов метода экземпляра
Example.ClassMethod();           // Вызов метода класса
```

```
// Следующие строки не компилируются:
example.ClassMethod();          // Обращение к методу класса
                                // через экземпляр
Example.InstanceMethod();       // Обращение к методу экземпляра
                                // через класс
```


ЗАПОМНИ!

Каждый экземпляр класса имеет собственную, закрытую копию любых членов экземпляра. Но все экземпляры одного и того же класса имеют одни и те же члены класса (как члены-данные, так и методы) и их значения.

Выражение `example.InstanceMethod()` передает управление коду, содержащемуся внутри метода. Процесс вызова `Example.ClassMethod()` практически такой же. В результате выполнения приведенного выше фрагмента кода (после того как будут закомментированы не компилирующиеся строки) на экран выводится следующее:

```
Это метод экземпляра
Это метод класса
```


ЗАПОМНИ!

После того как метод завершает свою работу, он передает управление в точку, из которой был вызван. Таким образом, управление передается инструкции, следующей за инструкцией вызова.

В приведенном примере код методов не делает ничего особенного, кроме вывода на экран единственной строки, но в общем случае методы выполняют различные сложные операции, такие как вычисление математических функций, объединение строк, сортировка массивов или отправка электронных писем. Словом, сложность решаемых методами задач ничем не ограничена. Методы могут быть любого размера и любой степени сложности, но все же лучше, чтобы они были небольшими по размеру для удобства работы с ними и уменьшения вероятности ошибок.


СОВЕТ

Эта книга при описании методов в тексте — как в случае `InstanceMethod()` — содержит пару скобок `()`, чтобы было легче распознать, что речь идет о методе. В противном случае читатель может запутаться, пытаясь разобраться в тексте.

Использование методов в ваших программах

В этом разделе для демонстрации того, как разумное определение методов может сделать программу проще для написания и понимания, будет взята монолитная программа `CalculateInterestTable` из главы 5, “Управление потоком выполнения”, и разделена на несколько методов, что делает программу

более легкой для написания и понимания. Такой процесс переделки рабочего кода при сохранении его функциональности называется *рефакторингом*, и Visual Studio 2012 и более поздние обеспечивают удобное меню рефакторинга Refactor, которое автоматизирует большинство распространенных задач рефакторинга. При работе с Visual Studio выберите пункт меню Edit⇨Refactor (Правка⇨Рефакторинг), чтобы получить доступ к возможностям рефакторинга.

ИСПОЛЬЗОВАНИЕ КОММЕНТАРИЕВ

Чтение комментариев *при опущенном программном коде* должно способствовать пониманию намерений программиста. Если это не так, значит, вы плохо комментируете свои программы. И наоборот: если вы не можете, опустив большинство комментариев, понять, что делает программа, на основании имен методов, значит, вы недостаточно ясно именуете методы и/или делаете их слишком большими. Меньшие методы предпочтительнее, а хорошие имена методов предпочтительнее комментариев. (Вот почему реальный код имеет гораздо меньше комментариев, чем примеры кода в этой книге. В коде книги комментарии используются для более подробного объяснения.)

ЗАПОМНИ!

Точную информацию об определениях и вызовах методов вы найдете в последующих разделах этой главы. Данный пример — не более чем просто обзор. "Скелет" программы `CalculateInterestTable` выглядит следующим образом:

```
public static void Main(string[] args)
{
    // Приглашение ввести начальный вклад.
    // Если вклад отрицателен, генерируется сообщение об ошибке.
    // Приглашение для ввода процентной ставки.
    // Если процентная ставка отрицательна, генерируется
    // сообщение об ошибке.
    // Приглашение для ввода количества лет.
    // Вывод введенных данных.
    // Цикл по введенному количеству лет.
    while(year <= duration)
    {
        // Вычисление значения вклада с начисленными процентами.
        // Вывод результата вычислений.
    }
}
```

Это пример хорошего метода проектирования методов. Если вы изучите программу, то увидите, что она состоит из следующих трех частей:

- часть начального ввода данных, в которой пользователи вводят вклад, процентную ставку и срок;
- раздел, выводящий введенную информацию на экран, чтобы пользователь мог убедиться в корректности ввода;
- последняя часть кода, создающая и выводящая таблицу на экран.

Это хорошее начало для выполнения рефакторинга. Кроме того, внимательнее рассмотрев часть ввода начальной информации, вы увидите, что код для ввода

- вклада,
- процентной ставки и
- срока

практически один и тот же.

Это наблюдение дает еще одну точку для рефакторинга. Кроме того, вы можете написать пустые методы для некоторых из этих комментариев, а затем заполнить их один за другим. Этот подход называется *программированием по намерению* (programming by intention). Таким образом можно получить новую версию программы — `CalculateInterestTableWithMethods`:

```
using System;
// Генерация таблицы роста вклада по тому же алгоритму, что
// и в ранее рассматривавшихся программах, однако в этой
// программе работа распределена между несколькими
// методами.
namespace CalculateInterestTableWithMethods {
    public class Program
    {
        public static void Main(string[] args) {
            // Раздел 1 - ввод данных для создания таблицы
            decimal principal = 0M;
            decimal interest  = 0M;
            decimal duration  = 0M;
            InputInterestData(ref principal, ref interest,
                              ref duration);

            // Раздел 2 - проверка введенных данных путем вывода
            // их пользователю на экран
            Console.WriteLine();     // Пропуск строки
            Console.WriteLine("Вклад            = " +
                              principal);
            Console.WriteLine("Процентная ставка = " +
                              interest + "%");
            Console.WriteLine("Срок             = " +
                              duration + " лет");
```

```
    Console.WriteLine();

    // Раздел 3 - вывод таблицы вкладов по годам
    OutputInterestTable(principal, interest, duration);

    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}

// InputInterestData - ввод с клавиатуры вклада,
// процентной ставки и срока для расчета таблицы
// Этот метод реализует раздел 1, разбивая его на три
// компонента
public static void InputInterestData(ref decimal principal,
                                     ref decimal interest,
                                     ref decimal duration) {
    // 1а Получение вклада
    principal = InputPositiveDecimal("вклад");

    // 1б Получение процентной ставки
    interest = InputPositiveDecimal("процентная ставка");

    // 1в Получение срока
    duration = InputPositiveDecimal("срок");
}

// InputPositiveDecimal возвращает положительное число
// типа decimal, введенное с клавиатуры
// Выполняется только одна проверка - на
// неотрицательность введенного значения
public static decimal InputPositiveDecimal(string prompt) {
    // Цикл выполняется, пока не будет введено верное
    // значение
    while (true) {
        // Приглашение для ввода
        Console.Write("Введите " + prompt + ":");

        // Получение значения типа decimal с клавиатуры
        string input = Console.ReadLine();
        decimal value = Convert.ToDecimal(input);

        // Выход из цикла при вводе корректного значения
        if (value >= 0)
        {
          // Возврат введенного значения
          return value;
        }

        // В противном случае генерируется и выводится
        // сообщение об ошибке
```

```
            Console.WriteLine(prompt +
                " не может иметь отрицательное значение");
            Console.WriteLine("Попробуйте еще раз");
            Console.WriteLine();
        }
    }

    // OutputInterestTable для заданных значений вклада,
    // процентной ставки и срока генерирует и выводит на
    // экран таблицу роста вклада
    // Реализация раздела 3 основной программы
    public static void OutputInterestTable(decimal principal,
                                           decimal interest,
                                           decimal duration) {
        for (int year = 1; year <= duration; year++)
        {
            // Вычисление начисленных процентов
            decimal interestPaid;
            interestPaid = principal * (interest / 100);

            // Вычисление значения нового вклада путем
            // добавления начисленных процентов к основному
            // вкладу
            principal = principal + interestPaid;

            // Округление вклада до копеек
            principal = decimal.Round(principal, 2);

            // Вывод результата
            Console.WriteLine(year + "-" + principal);
        }
    }
}
}
```

Раздел `Main()` состоит из трех очевидных частей, каждая из которых снабжена комментарием, выделенным полужирным шрифтом. Кроме того, раздел 1, в свою очередь, поделен на три подраздела — 1а, 1б и 1в.

Вам не следует пытаться выделять в своих исходных текстах комментарии полужирным шрифтом или указывать номера разделов. Исходный текст реальной программы — и без того сложная и запутанная штука, чтобы вносить в него искусственные усложнения. На практике для понимания достаточно ясных и информативных имен методов, указывающих их назначение.

В разделе 1 для ввода значений трех переменных, необходимых для работы программы (`principal`, `interest` им `duration`), вызывается метод `Input InterestData()`. В разделе 2 полученные значения выводятся на экран так же, как и в предыдущих версиях программы. В разделе 3 строится и выводится на экран таблица вкладов с помощью метода `OutputInterestTable()`.

Начнем с конца, с метода `OutputInterestTable()`. В нем содержится цикл, в котором выполняется вычисление начисленных процентов, точно так, как в программе `CalculateInterestTable` без методов. Преимущество данной версии заключается в том, что при разработке этой части кода не нужно сосредоточиваться на деталях ввода и верификации данных. При написании этого метода следует просто думать о том, как вычислить и вывести таблицу для уже полученных значений. После выполнения метода управление вернется в строку, следующую за вызовом метода `OutputInterestTable()`. `OutputInterestTable()` — хороший повод для того, чтобы воспользоваться меню рефакторинга Refactor в Visual Studio. Для этого выполните следующие действия.

1. **Воспользуйтесь в качестве стартовой точки примером `CalculateInterestTableMoreForgiving` из главы 5, "Управление потоком выполнения", выбрав исходный текст от объявления переменной `year` до конца цикла `while`:**

```
int year = 0;              // Переменная цикла
while(year <= duration) // и весь цикл while
{
    //...
}
```

2. **Выберите команду меню Edit⇨Refactor⇨Extract Method.**
3. **В диалоговом окне Rename: New Method введите `OutputInterestTable`.**

   Обратите внимание, что каждое местоположение в коде, где есть ссылка на новый метод, при вводе автоматически изменяется. Предложенная сигнатура нового метода начинается с ключевых слов `private static` и включает в себя `principal`, `interest` и `duration` в скобках. (В части 1, "Основы программирования на C#", книги в качестве альтернативы ключевому слову `public` вводится `private`. Пока что, если захотите, вы сможете сделать метод открытым (`public`) после рефакторинга.)

```
private static decimal OutputInterestTable(decimal principal,
                                           decimal interest,
                                           int duration)
```

4. **Щелкните на кнопке Apply для завершения рефакторинга.**

   Выбранный вами в п. 1 код располагается после `Main()` и именуется `OutputInterestTable()`. На месте, где он находился ранее, вы увидите вызов этого метода:

```
principal = OuputInterestTable(principal, interest, duration);
```

Могут применяться и другие виды рефакторинга — например, изменение порядка параметров метода, но дальнейшее углубление в эту тему выходит за рамки данной книги.


СОВЕТ

Поскольку C# поддерживает именованные параметры, вы можете указывать параметры метода в любом порядке, предваряя значение параметра его именем при вызове метода. Подробнее об этом вы узнаете немного позже, а пока просто знайте, что в C# 4.0 и более поздних версиях переупорядочение параметров проблемой не является.

В методе `InputInterestData()` вы сосредоточиваетесь только на вводе трех значений типа `decimal`. В данном случае, несмотря на три различные переменные, действия по их вводу идентичны и могут быть размещены в методе `InputPositiveDecimal()`, который одинаково применим как для ввода вклада, так и для ввода процентной ставки и срока, для которого выполняется расчет. Заметьте, что три цикла `while` в исходной программе превратились в один в теле метода `InputPositiveDecimal()`. Тем самым устранено дублирование кода, которое всегда нежелательно.

Метод `InputPositiveDecimal()` выводит приглашение и ожидает ввода пользователя. Если введенное пользователем значение неотрицательно, он возвращает его вызвавшему его методу. Если же введенное значение отрицательно, метод выводит сообщение об ошибке и повторяет цикл ввода. С точки зрения пользователя, программа работает точно так же, как и раньше:

```
Введите вклад: 100
Введите процентную ставку: -10
Процентная ставка не может быть отрицательной
Попробуйте еще раз

Введите процентную ставку: 10
Введите срок: 10

Вклад             = 100
Процентная ставка = 10%
Срок              = 10 лет

1-110.0
2-121.00
3-133.10
4-146.41
5-161.05
6-177.16
7-194.88
8-214.37
9-235.81
10-259.39
Нажмите <Enter> для завершения программы...
```

ЗАЧЕМ БЕСПОКОИТЬСЯ О МЕТОДАХ?

Когда в 1950-е годы в Фортране появилась концепция функции, ее единственной целью было избежать дублирования кода. Предположим, вы пишете программу, которая должна вычислять отношение двух чисел во многих местах. В этом случае программа может просто вызывать в этих местах метод `CalculateRatio()`, позволяющий избежать дублирования кода. Такая экономия может показаться не слишком большой, если метод состоит всего из пары строк, но методы бывают разными; они могут быть очень сложными и большими. Кроме того, распространенные методы наподобие `WriteLine()` могут использоваться в сотнях различных мест.

Второе преимущество применения методов также очевидно: проще корректно написать и отладить один метод, чем десяток фрагментов кода, и вдвойне проще сделать это, если метод невелик. Метод `CalculateRatio()` включает проверку того, что знаменатель в отношении не равен нулю. Если у вас имеется множество фрагментов кода, а не один метод, то, скорее всего, в некоторых местах программы вы просто забудете вставить эту проверку.

Менее очевидно третье преимущество: хорошо спроектированные методы снижают сложность программы. Каждый метод должен соответствовать некоторой концепции. Вы должны быть способны указать назначение каждого метода без использования слов *и* и *или*. Вы должны следовать принципу "один метод — одна задача".

Метод наподобие `calculateSin()` служит идеальным примером. Программист, реализуя сложные вычисления, совершенно не должен беспокоиться, как именно будут применены их результаты. Прикладной программист может использовать метод `calculateSin()`, не интересуясь, как именно он устроен и работает. Этот подход существенно снижает количество вещей, о которых должен помнить прикладной программист. Большую работу гораздо проще сделать, если разделить ее на части.

Большие программы, как, например, текстовый редактор, строятся из множества методов разного уровня абстракции. Например, метод `RedisplayDocument()` должен вызывать метод `Reparagraph()` для вывода абзацев документа. Этот метод, в свою очередь, должен вызывать метод `CalculateWordWrap()` для вычисления длин отдельных строк абзаца. Метод `CalculateWordWrap()` может вызывать метод `LookUpWordBreak()`, определяющий, как должно быть разбито для переноса слово, стоящее в конце строки. Каждый из перечисленных методов решает одну задачу, которую можно сформулировать простым предложением (кстати, обратите внимание и на информативность названий методов).

Без возможности *абстрагирования* сложных концепций написание программы даже средней сложности становится практически нереализуемым, не говоря уже о создании операционных систем, игр, офисного программного обеспечения и тому подобных больших и сложных программ.

Аргументы метода

Метод, подобный приведенному ниже, полезен примерно так же, как и зубная щетка, которой может пользоваться только один человек. Это связано с тем, что никакие данные приведенному методу не передаются и им не возвращаются:

```
public static void Output()
{
    Console.WriteLine("Это метод");
}
```

Сравним этот пример с реальными методами. Например, метод вычисления синуса требует определенных входных данных (в конце концов, вы ведь вычисляете синус чего-то?). Аналогично при конкатенации двух строк нужно передать методу две строки и получить от метода результаты его работы. Следовательно, возникает крайняя необходимость в механизме обмена информацией с методом.

Передача аргументов методу

Значения, передаваемые методу, называются *аргументами метода* (другое часто используемое название — *параметры*). Большинство методов требуют для работы аргументов определенного типа. Вы передаете аргументы методу, перечисляя их в скобках после его имени. Проанализируем следующее небольшое дополнение к рассматривавшемуся ранее классу `Example`:

```
public class Example
{
    public static void Output(string someString)
    {
        Console.WriteLine("Метод Output() получил аргумент: "
                          + someString);
    }
}
```

Этот метод можно вызвать в самом классе следующим образом:

```
Output("Hello");
```

В результате можно получить вывод на экран:

```
Метод Output() получил аргумент: Hello
```

Программа передает методу `Output()` ссылку на строку `"Hello"`. Метод получает эту строку и присваивает ей имя `someString`. В теле метода `Output()` переменная `someString` может использоваться точно так, как любая другая переменная типа `string`.

Можно немного изменить пример:

```
string myString = "Hello";
Output(myString);
```

В этом фрагменте переменной `myString` присваивается ссылка на строку `"Hello"`. Вызов `Output(myString)` передает методу объект, на который ссылается переменная `myString`, т.е. ту же строку `"Hello"`, что и ранее. Этот процесс изображен на рис. 13.1. Результат работы фрагмента исходного текста тот же, что и до внесения в него изменений.

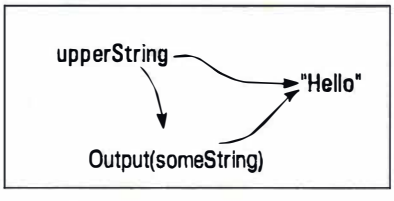


Рис. 13.1. Копирование значения `myString` в `someString`

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Заполнители, которые вы указываете для аргументов при написании метода (например, `someString` в `Output()`), являются *параметрами*. Значения, которые вы передаете методу через параметр, являются *аргументами*. В этой книге данные термины используются более или менее взаимозаменяемо.

Похожая идея — передача аргументов программе. Например, вы могли заметить, что `Main()` обычно принимает в качестве аргумента массив.

Передача методу нескольких аргументов

Можно определить метод с несколькими аргументами различных типов. Рассмотрим в качестве примера метод `AverageAndDisplay()`:

```
// AverageAndDisplay
using System;

namespace Example
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Обращение к методу-члену
            AverageAndDisplay("оценки 1", 3.5, "оценки 2", 4.0);

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
```

```
            Console.Read();
        }

        // AverageAndDisplay усредняет два числа и выводит
        // результат с использованием переданных меток
        public static void AverageAndDisplay(string s1, double d1,
                                              string s2, double d2)
        {
            double average = (d1 + d2) / 2;
            Console.WriteLine("Среднее "  + s1
                              + ", равной " + d1
                              + ", и "      + s2
                              + ", равной " + d2
                              + ", равно "  + average);
        }
    }
}
```

Вот как выглядит вывод этой программы на экран:

```
Среднее оценки 1, равной 3.5, и оценки 2, равной 4, равно 3.75
Нажмите <Enter> для завершения программы...
```

Метод `AverageAndDisplay()` объявлен с несколькими аргументами в том порядке, в котором они в нее передаются.

Как обычно, выполнение программы начинается с первой инструкции в `Main()`. Первая строка `Main()`, не являющаяся комментарием, вызывает метод `AverageAndDisplay()`, передавая ему две строки и два значения типа `double`.

Метод `AverageAndDisplay()` вычисляет среднее переданных значений типа `double`, `d1` и `d2`, переданных в метод вместе с их именами (содержащимися в переменных `s1` и `s2`), и сохраняет полученное значение в переменной `average`.

СОВЕТ

Изменение значений аргументов внутри метода может привести к ошибкам. Разумнее присвоить эти значения временным переменным и модифицировать уже их.

Соответствие определений аргументов их использованию

Каждый аргумент в вызове метода должен соответствовать определению метода как в смысле типа, так *и в смысле порядка*. Приведенный далее исходный текст некорректен и вызывает ошибку в процессе компиляции.

```
// AverageWithCompilerError - эта версия не компилируется!
using System;

namespace Example
{
    public class Program
    {
        public static void Main(string[] args)
```

```
        {
            // Обращение к методу-члену
            AverageAndDisplay("оценки 1", "оценки 2", 3.5, 4.0);

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }

        // AverageAndDisplay усредняет два числа и выводит
        // результат с использованием переданных меток
        public static void AverageAndDisplay(string s1, double d1,
                                              string s2, double d2)
        {
            double average = (d1 + d2) / 2;
            Console.WriteLine("Среднее "  + s1
                              + ", равной " + d1
                              + ", и "      + s2
                              + ", равной " + d2
                              + ", равно "  + average);
        }
    }
}
```

C# обнаруживает несоответствие типов передаваемых методу аргументов с аргументами в определении метода. Строка `"оценки 1"` соответствует типу `string` в определении метода; однако согласно определению метода вторым аргументом должно быть число типа `double`, в то время как при вызове вторым аргументом метода оказывается строка `string`.

Легко увидеть, что в коде переставлены местами второй и третий аргументы. Чтобы решить проблему, поменяйте их местами, чтобы они находились в правильном порядке.

Перегрузка методов

СОВЕТ

В одном классе может быть два метода с одним и тем же именем — *при условии различия их аргументов*. Это явление называется *перегрузкой* (overloading) имени метода.

```
// AverageAndDisplayOverloaded демонстрирует возможность
// перегрузки метода вычисления и вывода среднего значения
using System;

namespace AverageAndDisplayOverloaded
{
    public class Program
    {
        public static void Main(string[] args)
        {
```

```
            // Вызов первого метода-члена
            AverageAndDisplay("моей оценки", 3.5,
                              "твоей оценки", 4.0);
            Console.WriteLine();
            // Вызов второго метода-члена
            AverageAndDisplay(3.5, 4.0);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }

        // AverageAndDisplay - вычисление среднего значения двух
        // чисел и его вывод на экран с переданными методу
        // метками этих чисел
        public static
        void AverageAndDisplay(string s1, double d1,
                               string s2, double d2)
        {
            double average = (d1 + d2) / 2;
            Console.WriteLine("Среднее "  + s1
                              + ", равной " + d1);
            Console.WriteLine("и "        + s2
                              + ", равной " + d2
                              + ", равно "  + average);
        }
        public static void AverageAndDisplay(double d1,
                                              double d2)
        {
            double average = (d1 + d2) / 2;
            Console.WriteLine("среднее " + d1
                              + " и "      + d2
                              + " равно "  + average);
        }
    }
}
```

В программе определены две версии метода `AverageAndDisplay()`. Программа вызывает одну из них после другой, передавая им соответствующие аргументы. C# в состоянии определить по переданным методу аргументам, какую из версий следует вызвать, сравнивая типы передаваемых значений с определениями методов. Программа корректно компилируется и выполняется, выводя на экран следующие строки:

```
Среднее моей оценки, равной 3.5,
и твоей оценки, равной 4, равно 3.75

Среднее 3.5 и 4 равно 3.75
Нажмите <Enter> для завершения программы...
```

Вообще говоря, C# не позволяет иметь в одной программе два метода с одинаковыми именами. В конце концов, как тогда он сможет разобраться, какой из методов следует вызывать? Но дело в том, что в C# имя метода во внутреннем представлении компилятора включает не только имя метода, но и количество и типы его аргументов. Поэтому C# в состоянии различить методы

- `AverageAndDisplay(string,double,string,double)` и
- `AverageAndDisplay(double,double)`

Если рассматривать эти методы с их аргументами, становится очевидным, что они разные.

Реализация аргументов по умолчанию

В некоторых случаях для упрощения использования методу требуется предопределенное значение — *аргумент по умолчанию*. Если большинству разработчиков, использующих метод, требуется определенное значение, значение по умолчанию имеет смысл. В этом случае метод обладает достаточной гибкостью — разработчики, которым нужны значения, отличные от значения по умолчанию, по-прежнему имеют возможность их предоставления. Зачастую желательно иметь две (или более) версии метода. Для этого обычно используются два общеупотребительных способа.

- Один из методов представляет собой более сложную версию, обеспечивающую большую гибкость, но требующую большого количества аргументов от вызывающей программы, причем некоторые из них могут быть просто непонятны пользователю.


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Под *пользователем* метода подразумевается программист, применяющий его в своих программах, так что пользователь метода и пользователь готовой программы — это разные люди. Еще один термин, применяемый для обозначения такого рода пользователя, — *клиент*.

- Для некоторых аргументов предоставляются аргументы по умолчанию.

Аргументы по умолчанию легко реализовать с помощью перегрузки методов. Рассмотрим следующую пару методов `DisplayRoundedDecimal()`:

```
// MethodsWithDefaultArguments - две версии одного и того же
// метода, причем одна из них представляет версию второй с
// использованием значений аргументов по умолчанию

using System;
```

```
namespace MethodsWithDefaultArguments
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Вызов метода-члена
            Console.WriteLine("{0}",
                              DisplayRoundedDecimal(12.345678M, 3));
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }

        // DisplayRoundedDecimal преобразует значение типа
        // decimal в строку с определенным количеством значащих
        // цифр
        public static
        string DisplayRoundedDecimal(decimal value,
                                     int nNumberOfSignificantDigits)
        {
            // Сначала округляем число до указанного количества
            // значащих цифр ...
            decimal mRoundedValue =
                decimal.Round(value,
                              nNumberOfSignificantDigits);
            // ... и преобразуем его в строку
            string s = Convert.ToString(mRoundedValue);
            return s;
        }
        public static string DisplayRoundedDecimal(decimal value)
        {
            // Вызываем DisplayRoundedDecimal(decimal, int) с
            // указанием количества значащих цифр по умолчанию
            string s = DisplayRoundedDecimal(value, 2);
            return s;
        }
    }
}
```

Метод `DisplayRoundedDecimal(decimal,int)` преобразует значение типа `decimal` в значение типа `string` с определенным количеством значащих цифр после десятичной точки. Поскольку числа типа `decimal` часто применяются в финансовых расчетах, наиболее распространенными будут вызовы этого метода со вторым аргументом, равным 2. В анализируемой программе это предусмотрено, и вызов `DisplayRoundedDecimal(decimal)` с одним аргументом округляет значение этого аргумента до двух цифр после десятичной точки, позволяя пользователю не беспокоиться о смысле и числовом значении второго аргумента метода.

ЗАПОМНИ!

Обратите внимание на то, что версия метода `DisplayRoundedDecimal(decimal)` в действительности вызывает метод `DisplayRoundedDecimal(decimal,int)`. Такая практика позволяет избежать ненужного дублирования кода. Обобщенная версия метода может использовать существенно большее количество аргументов, которые ее разработчик может даже не включить в документацию.

ВНИМАНИЕ!

Необходимость излишне часто обращаться к справочной системе и документации, чтобы узнать значения аргументов по умолчанию, отвлекает программиста от основной работы, что делает ее более трудной, требующей больше времени и увеличивает вероятность появления ошибок.

Аргументы по умолчанию не просто сберегают силы ленивого программиста. Программирование — работа, требующая высочайшей степени концентрации, и излишние аргументы метода, для выяснения назначения и рекомендуемых значений которых необходимо обращаться к документации, затрудняют программирование, приводят к перерасходу времени и повышают вероятность внесения ошибок в код. Автор метода хорошо понимает взаимосвязи между аргументами метода и способен обеспечить несколько корректных перегруженных версий, более дружественных к клиенту.

Программисты на Visual Basic и C/C++ привыкли предоставлять значение по умолчанию для параметра непосредственно в сигнатуре метода. До выхода в свет C# 4.0 такое было невозможно в C#. Теперь эта возможность есть и у программистов на C#.

Например, хотя перегрузка метода в предыдущем примере является вполне приемлемым способом реализации параметра по умолчанию, можно также задать параметры по умолчанию, используя знак равенства (=):

```
// MethodsWithDefaultArguments2 - предоставление необязательного
// значения параметра метода для избежания перегрузки.
using System;

namespace MethodsWithDefaultArguments2
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Вызов метода-члена
            Console.WriteLine("{0}",
                              DisplayRoundedDecimal(12.345678M, 3));
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
```

```
        // DisplayRoundedDecimal преобразует значение типа
        // decimal в строку с определенным количеством значащих
        // цифр. Этот параметр необязателен. При вызове метода без
        // второго параметра используется значение по умолчанию.
        public static string DisplayRoundedDecimal(
                        decimal value,
                        int nNumberOfSignificantDigits = 2)
        {
            // Сначала округляем число до указанного количества
            // значащих цифр ...
            decimal mRoundedValue =
                decimal.Round(value,
                              nNumberOfSignificantDigits);
            // ... и преобразуем его в строку
            string s = Convert.ToString(mRoundedValue);
            return s;
        }
    }
}
```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Почему Microsoft внесла эти изменения? Ответ — COM. Объектная модель компонентов (Component Object Model — COM) была архитектурной парадигмой для продуктов Microsoft до выпуска .NET и до сих пор широко распространена. Office, например, полностью разработан с использованием COM. Приложения COM разрабатываются на C ++ или Visual Basic 6 и более ранних версиях, а методы из этих классов допускают необязательные параметры. Таким образом, связь с COM без использования дополнительных параметров может стать затруднительной. Чтобы устранить этот дисбаланс, в C# 4.0 были добавлены необязательные параметры (наряду с рядом других функциональных возможностей). В главе 21, "Именованные и необязательные параметры", именованные и необязательные параметры рассматриваются более подробно.

Возврат значений из метода

Многие реальные операции создают значения, которые должны быть возвращены тому, кто вызвал эти операции. Например, метод `sin()` получает аргумент и возвращает значение тригонометрической функции "синус" для данного аргумента. Метод может вернуть значение вызывающему методу двумя способами. Наиболее распространенный — с помощью команды `return`; при втором способе используются возможности *передачи аргументов по ссылке*.

Возврат значения оператором return

В приведенном далее фрагменте исходного текста демонстрируется небольшой метод, возвращающий среднее значение переданных ему аргументов:

```
public class Example
{
    public static double Average(double d1, double d2)
    {
        double average = (d1 + d2) / 2;
        return average;
    }
    public static void Test()
    {
        double v1 = 1.0;
        double v2 = 3.0;
        double averageValue = Average(v1, v2);
        Console.WriteLine("Среднее для " + v1
                          + " и " + v2 + " равно "
                          + averageValue);
        // Такой метод также вполне работоспособен
        Console.WriteLine("Среднее для " + v1
                          + " и " + v2 + " равно "
                          + Average(v1, v2));
    }
}
```

Прежде всего, обратите внимание на то, что метод объявлен как `public static double Average()`: тип `double` перед именем метода указывает на тот факт, что метод `Average()` возвращает вызывающему методу значение типа `double`. Метод `Average()` использует имена `d1` и `d2` для значений, переданных ему в качестве аргументов. Он создает переменную `average`, которой присваивает среднее значение этих переменных. Затем значение, содержащееся в переменной `average`, возвращается вызывающему методу.

ВНИМАНИЕ!

Программисты иногда говорят, что "метод возвращает `average`". Это некорректное выражение. Говорить, что передается или возвращается `average` или иная переменная, — неточно. В данном случае вызывающему методу возвращается *значение*, содержащееся в переменной `average`.

Вызов `Average()` из метода `Test()` выглядит так же, как и вызов любого другого метода; однако значение типа `double`, возвращаемое методом `Average()`, сохраняется в переменной `averageValue`.

ЗАПОМНИ!

Метод, который возвращает значение (как, например, `Average()`), не может завершиться просто по достижении закрывающей фигурной скобки, поскольку C# совершенно непонятно, какое же именно значение должен будет вернуть этот метод? Для этого обязательно наличие оператора `return`.

Определение метода без возвращаемого значения

Выражение `public static double Average(double,double)` объявляет метод `Average()` как возвращающий значение типа `double`. Однако существуют методы, не возвращающие ничего. Ранее вы сталкивались с примером такого метода, `AverageAndDisplay()`, который выводил вычисленное среднее значение на экран, ничего не возвращая вызывающему методу. Вместо того чтобы опустить в объявлении такого метода тип возвращаемого значения, в C# указывается `void`:

```
public void AverageAndDisplay(double, double)
```

Ключевое слово `void`, употребленное вместо имени типа, по сути, означает *отсутствие типа*, т.е. указывает, что метод `AverageAndDisplay()` ничего не возвращает вызывающему методу. (В C# любое объявление метода обязано указывать возвращаемый тип, даже если это `void`.)


ЗАПОМНИ!

Метод, который не возвращает значения, программистами называется *void-методом*, по использованному ключевому слову в его описании.

Методы, не являющиеся `void`-методами, возвращают управление вызывающему методу при выполнении оператора `return`, за которым следует возвращаемое вызывающему методу значение. Поскольку `void`-метод не возвращает никакого значения, выход из него осуществляется посредством оператора `return` без какого бы то ни было значения либо при достижении закрывающей тело метода фигурной скобки. Рассмотрим следующий метод `DisplayRatio()`:

```
public class Example
{
    public static void DisplayRatio(double numerator,
                                     double denominator)
    {
        // Если знаменатель равен 0...
        if (denominator == 0.0)
        {
            // ... вывести сообщение об ошибке и вернуть
            // управление вызывающему методу...
            Console.WriteLine("Знаменатель не может быть нулем");
            // Выход из метода
            return;
        }
        // Эта часть метода выполняется только в том случае,
        // когда знаменатель не равен нулю
        double ratio = numerator / denominator;
        Console.WriteLine("Отношение " + numerator
                          + " к " + denominator
                          + " равно " + ratio);
    } // Если знаменатель не равен нулю, выход из метода
}     // выполняется здесь
```

МЕТОД WriteLine()

Вы могли заметить, что метод `WriteLine()`, использовавшийся в рассматриваемых программах, представляет собой не более чем вызов метода класса `Console`:

```
Console.WriteLine("Это — вызов метода");
```

Метод `WriteLine()` — один из множества предопределенных методов, предоставляемых библиотекой .NET. `Console` — предопределенный класс, предназначенный для использования в консольных приложениях.

Аргументом метода `WriteLine()`, применявшимся в рассмотренных выше примерах, является строка `string`. Оператор `+` позволяет программисту собрать эту строку из нескольких строк или строк и переменных встроенных типов, например, так:

```
string s = "Маша";
Console.WriteLine("Меня зовут " + s +
                  ", и мне " + 3 + " года");
```

В результате вы увидите выведенную на экран строку "`Меня зовут Маша, и мне 3 года`".

Второй вид метода `WriteLine()` допускает наличие более гибкого множества аргументов, например:

```
Console.WriteLine("Меня зовут {0} и мне {1} года",
                  "Маша", 3);
```

Первый аргумент такого вызова называется форматной строкой. В данном примере строка `"Маша"` вставляется вместо символов `{0}` — нуль указывает на первый аргумент после командной строки. Целое число 3 вставляется в позицию, помеченную как `{1}`. Этот вид метода более эффективен, поскольку конкатенация строк не так проста, как это звучит, и не столь эффективна.

Метод `DisplayRatio()` начинает работу с проверки, не равно ли значение `denominator` нулю.

- Если значение `denominator` равно нулю, программа выводит сообщение об ошибке и возвращает управление вызывающему методу, не пытаясь вычислить значение отношения. При попытке вычислить отношение произошла бы ошибка деления на нуль с аварийным остановом программы в результате.
- Если значение `denominator` не равно нулю, программа выводит на экран значение отношения. При этом закрывающая фигурная скобка после вызова метода `WriteLine()` является закрывающей скобкой метода `DisplayRatio()` и, таким образом, представляет собой точку возврата из метода в вызывающую программу.

Если бы это было единственной разницей, не о чем было бы много писать. Однако вторая форма `WriteLine()` предоставляет ряд элементов управления форматом вывода, описанных в главе 3, "Работа со строками".

Возврат нескольких значений с использованием кортежей

В версиях C# до C# 7.0 каждое возвращаемое значение является отдельным объектом. Это может быть сложный объект, но это все еще единственный объект. В C# 7.0 можно возвращать из методов несколько значений, используя кортежи. Кортеж — это разновидность динамического массива, номинально содержащего два элемента, которые можно интерпретировать как пару "ключ–значение" (но это не обязательно). В C# вы также можете создавать кортежи, содержащие более двух элементов. Многие языки, такие как Python, используют кортежи, чтобы упростить кодирование и значительно облегчить работу со значениями.

Фактически C# 4.*x* ввел концепцию кортежей как часть подхода динамического программирования. Однако C# 7.0 продвигает использование кортежей, позволяющих возвращать несколько значений, а не только один объект. Подробный обзор кортежей выходит за рамки данной книги, но они так хорошо работают при возврате сложных данных, что вам, определенно, необходимо знать кое-что об этом их применении.

Кортеж с двумя элементами

Кортеж основан на типе данных `Tuple`, который может принимать одно или два входных данных, причем наиболее распространенными являются кортежи с двумя элементами (в противном случае вы можете просто вернуть единственный объект). Лучший способ работы с кортежами — предоставить типы данных переменных, с которыми вы планируете работать, как часть объявления. Вот пример метода, который возвращает кортеж:

```
static Tuple<string, int> getTuple()
{
    // Возвращает единое значение как кортеж
    return new Tuple<string, int>("Hello", 123);
}
```

Код начинается с указания того, что `getTuple()` возвращает кортеж `Tuple` из двух элементов с типами `string` и `int`. Ключевое слово `new` используется для создания экземпляра `Tuple`, с типами данных, описанными в угловых скобках, `<string,int>`, и указанными значениями данных. Метод `getTuple()`

возвращает два значения, с которыми вы можете работать по отдельности, как показано далее:

```
// Начало программы.
static void Main(string[] args)
{
    // Получение кортежа.
    Console.WriteLine(
            getTuple().Item1 + " " + getTuple().Item2);
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}
```

Чтобы получить доступ к элементам кортежа наподобие данного, вы вызываете `getTuple()`, добавляете точку и затем указываете, какой именно элемент использовать — `Item1` или `Item2`. Этот пример просто демонстрирует, как работают кортежи. Его вывод выглядит следующим образом:

```
Hello 123
Нажмите <Enter> для завершения программы...
```

СОВЕТ

Использование кортежа позволяет возвращать два значения, не прибегая к сложным типам данных или другим странным структурам. Это делает ваш код проще, когда требования к выходным данным вписываются в рамки кортежа. Например, при выполнении определенных математических операций может быть необходимо вернуть результат и остаток или вещественную и мнимую части комплексного числа.

Применение метода `Create()`

Альтернативный способ создания кортежа — использование метода `Create()`. Результат получается такой же, как и при работе с методом, описанным в предыдущем разделе. Вот пример использования метода `Create()`:

```
// Применение метода Create().
var myTuple = Tuple.Create<string, int>("Hello", 123);
Console.WriteLine(myTuple.Item1 + "\t" + myTuple.Item2);
```

Этот подход не так безопасен, как использование метода, показанного в предыдущем разделе, потому что элементы `myTuple` могут быть изменены извне. В конструкторе можно исключить часть `<string, int>`, так как компилятор может выяснить типы элементов `myTuple` из переданных входных данных.

Многоэлементные кортежи

Истинная ценность кортежа заключается в создании наборов данных с использованием чрезвычайно простых методов. Вы можете выбрать для просмотра `Item1` в качестве ключа и `Item2` в качестве значения. Многие типы наборов данных полагаются на парадигму "ключ–значение", и такая трактовка кортежа делает его невероятно полезным. В следующем примере показаны создание и возврат соответствующего набора данных.

```
static Tuple<string, int>[] getTuple()
{
    // Создание нового кортежа.
    Tuple<string, int>[] aTuple =
    {
        new Tuple<string, int>("One", 1),
        new Tuple<string, int>("Two", 2),
        new Tuple<string, int>("Three", 3)
    };
    // Возврат списка значений с использованием кортежа.
    return aTuple;
}
```

Как и в предыдущем разделе, вы указываете в качестве возвращаемого типа `Tuple`, но с добавлением пары квадратных скобок (`[]`), аналогичных тем, которые используются для массива. Квадратные скобки говорят C#, что эта версия `getTuple()` возвращает несколько кортежей, а не один.

Чтобы создать набор данных, вы начинаете с объявления переменной `aTuple`. Каждая новая запись в кортеже требует нового объявления кортежа с необходимыми входными данными. Все они помещаются в фигурные скобки. Чтобы вернуть кортеж, как обычно, используется оператор `return`.

Доступ к кортежу требует использования перечислителя, и вы можете делать все, что вы обычно делаете с перечислителем, например работать с отдельными значениями с помощью `foreach`:

```
static void Main(string[] args)
{
    // Получение набора кортежей.
    Tuple<string, int>[] myTuple = getTuple();
    // Вывод значений.
    foreach (var Item in myTuple)
    {
        Console.WriteLine(Item.Item1 + "\t" + Item.Item2);
    }
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}
```

Цикл `foreach` помещает отдельные элементы из `myTuple` в `Item`. Затем к элементам данных выполняется обращение с использованием `Item1` и `Item2`, как и ранее. Вот вывод этого примера:

```
One 1
Two 2
Three 3
Нажмите <Enter> для завершения программы...
```

Создание кортежей более чем с двумя элементами

Кортежи могут иметь от одного до восьми элементов. Чтобы получить более восьми элементов, восьмой элемент должен содержать еще один кортеж. Вложенные кортежи позволяют возвращать почти бесконечное количество элементов, но в какой-то момент действительно нужно остановиться, взглянуть на сложность своего кода и посмотреть, не стоит ли уменьшить количество возвращаемых элементов. В противном случае ваше приложение будет медленно работать и требовать много ресурсов. Вот пример версии кода из предыдущего раздела с использованием кортежей с тремя элементами:

```
static Tuple<string, int, bool>[] getTuple()
{
    // Создание нового кортежа.
    Tuple<string, int, bool>[] aTuple =
    {
        new Tuple<string, int, bool>("One", 1, true),
        new Tuple<string, int, bool>("Two", 2, false),
        new Tuple<string, int, bool>("Three", 3, true)
    };
    // Возврат списка значений с использованием кортежа.
    return aTuple;
}
```

Подход следует той же схеме, что и раньше. Разница лишь в том, что вы предоставляете для каждого кортежа больше значений. Не имеет значения, создаете ли вы один кортеж или массив, используемый в качестве набора данных. Любой выбор позволяет использовать до восьми элементов в кортеже.

Глава 14

Поговорим об этом

В ЭТОЙ ГЛАВЕ...

- Передача объекта в метод
- Методы классов и методы экземпляров
- Что такое `this`
- Работа с локальными функциями

После статических методов, рассматривавшихся в главе 13, "Методы", перейдем к нестатическим *методам* класса. Статические методы принадлежат всему классу, в то время как нестатические — экземплярам класса. Различие между статическими и нестатическими методами очень важно.

Передача объекта в метод

Ссылка на объект передается в метод точно так же, как и переменная, принадлежащая типу-значению, с единственным отличием — объекты всегда передаются в метод только по ссылке. Следующая маленькая программа продемонстрирует, каким образом можно передать объект методу:

```
// PassObject - демонстрация передачи объекта методу
using System;
```

```
namespace PassObject
{
    public class Student
    {
        public string name;
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            Student student = new Student();
            // Присваиваем имя путем непосредственного
            // обращения к полю объекта
            Console.WriteLine("Сначала:");
            student.name = "Madeleine";
            OutputName(student);
            // Изменяем имя с помощью метода
            Console.WriteLine("После изменения:");
            SetName(student, "Willa");
            OutputName(student);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }

        // OutputName - вывод имени студента
        public static void OutputName(Student student)
        {
            // Вывод текущего имени студента
            Console.WriteLine("Student.name = {0}",
                              student.name);
        }

        // SetName - изменение имени студента
        public static void SetName(Student student,
                                   string name)
        {
            student.name = name;
        }
    }
}
```

Программа создает объект `student`, в котором не содержится ничего, кроме имени. Сначала она присваивает имя непосредственно и передает объект `student` методу вывода `OutputName()`, который выводит его имя на консоль.

Затем программа изменяет имя `student` посредством метода `SetName()`. Поскольку все объекты в C# передаются в методы по ссылке, изменения, внесенные в объект `student` в методе, сохраняются и после возврата из него. Когда

метод Main() опять вызывает метод для вывода имени студента, последний выводит измененное имя, что видно из вывода программы на экран:

```
Сначала:
Student.name = Madeleine
После изменения:
Student.name = Willa
Нажмите <Enter> для завершения программы...
```

Метод SetName() может изменить имя в самом объекте типа Student и закрепить его за этим объектом.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Обратите внимание на то, что при передаче *ссылочного объекта* в метод ключевое слово ref не используется. Метод, которому объект передается по ссылке, может посредством этой *ссылки* изменить *содержимое* объекта, но не в состоянии присвоить новый объект, как показано в следующем фрагменте исходного текста:

```
Student student = new Student();
SetName(student, "Pam");
Console.WriteLine(student.name); // Все еще "Pam"
...
// Измененный метод SetName():
public static void SetName(Student student, string name)
{
    student = new Student(); // Не изменяет объект student
                             // вне SetName()
    student.Name = name;
}
```

Определение методов

Класс представляет собой набор элементов, описывающий объект или концепцию реального мира. Например, класс Vehicle может содержать данные о максимальной скорости, максимальном разрешенном весе, количестве пассажирских мест и т.д. Однако транспортное средство имеет и активные свойства — *поведение*: возможность тронуться с места, остановиться и т.п. Такие действия можно описать методами, работающими с данными транспортного средства. Эти методы представляют собой такую же часть класса Vehicle, как и его члены-данные.

Определение статического метода

Например, можно переписать программу из предыдущего раздела следующим образом:

```
// StudentClassWithMethods - демонстрация методов,
// работающих с данными внутри класса. Класс отвечает
// за свои данные и за работу с ними
using System;
namespace StudentClassWithMethods
{
    // Методы OutputName и SetName являются членами
    // класса Student, а не класса Program
    public class Student
    {
        public string name;
        // OutputName - вывод имени
        public static void OutputName(Student student)
        {
            // Выводим имя
            Console.WriteLine("Имя студента - {0}", student.name);
        }
        // SetName - модификация имени студента
        public static void SetName(Student student, string name)
        {
            student.name = name;
        }
    }
    public class Program
    {
        public static void Main(string[] args)
        {
            Student student = new Student();
            // Непосредственная установка имени
            Console.WriteLine("Сначала:");
            student.name = "Madeleine";
            Student.OutputName(student); // Метод класса Student
            Console.WriteLine("После:");
            // Изменение имени при помощи метода
            Student.SetName(student, "Willa");
            Student.OutputName(student);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

По сравнению с программой `PassObject` данная программа имеет только одно важное изменение: методы `OutputName()` и `SetName()` перенесены в класс `Student`. Из-за этого изменения метод `Main()` вынужден обращаться к означенным методам с указанием класса `Student`. Эти методы теперь являются членами класса `Student`, а не `Program`, которому принадлежит метод `Main()`.

Это маленький, но достаточно важный шаг. Размещение метода `OutputName()` в классе приводит к повышению степени повторного

использования: внешние методы, которым необходимо вывести объект на экран, могут найти метод `OutputName()` вместе с другими методами в классе, следовательно, писать такие методы для каждой программы, применяющей класс `Student`, не требуется.

Указанное решение лучше и с философской точки зрения. Класс `Program` не должен беспокоиться о том, как инициализировать имя объекта `Student` или вывести это имя на экран. Всю эту информацию должен содержать сам класс `Student`. *Объекты отвечают сами за себя*. Фактически метод `Main()` не должен инициализировать объект именем `"Madeleine"` непосредственно — в этом случае также следует использовать метод `SetName()`.

Внутри самого класса `Student` один метод-член может вызывать другой без явного указания имени класса. Метод `SetName()` может вызвать метод `OutputName()`, не указывая имени класса. Если имя класса не указано, C# считает, что вызван метод из того же класса.

Определение метода экземпляра

Хотя `OutputName()` и `SetName()` представляют собой статические методы, их легко сделать нестатическими, или методами *экземпляров*.


ЗАПОМНИ!

Все статические члены класса называются *членами класса*, все нестатические — *членами экземпляра*.

Обращение к членам данных объекта — *экземпляра* класса — выполняется посредством указания объекта, а не класса:

```
Student student = new Student();  // Создание экземпляра Student
student.name    = "Madeleine";    // Обращение к члену
```

Язык C# позволяет вызывать *нестатические* методы аналогично:

```
student.SetName("Madeleine");
```

Следующий пример демонстрирует это:

```
// InvokeMethod - вызов метода-члена с указанием объекта
using System;

namespace InvokeMethod
{
    class Student
    {
        // Информация об имени студента
        public string firstName;
        public string lastName;
```

```
        // SetName - сохранение информации об имени
        public void SetName(string fName, string lName)
        {
            firstName = fName;
            lastName  = lName;
        }

        // ToNameString преобразует объект класса Student в
        // строку для вывода
        public string ToNameString()
        {
            string s = firstName + " " + lastName;
            return s;
        }
    }

    public class Program
    {
        public static void Main()
        {
            Student student = new Student();
            student.SetName("Stephen", "Davis");
            Console.WriteLine("Имя студента - "
                              + student.ToNameString());
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

Вывод данной программы состоит из одной строки:

```
Имя студента - Stephen Davis
```

Эта программа очень похожа на программу `StudentClassWithMethods`. В приведенной версии используются *нестатические* методы для работы с именем и фамилией.

Программа начинает работу с создания нового объекта `student` класса `Student`, после чего вызывает метод `SetName()`, который сохраняет строки `"Stephen"` и `"Davis"` в членах-данных `firstName` и `lastName`. И наконец программа вызывает метод `ToNameString()`, возвращающий имя студента, составленное из двух строк.

Вернемся вновь к методу `SetName()`, предназначенному для изменения значений полей объекта класса `Student`. Какой именно объект модифицирует метод `SetName()`? Рассмотрим, как работает следующий пример:

```
Student christa = new Student(); // Создаем двух совершенно
Student sarah   = new Student(); // разных студентов
```

```
christa.SetName("Christa", "Smith");
sarah.SetName("Sarah", "Jones");
```

Первый вызов `SetName()` изменяет поля объекта `christa`, а второй — объекта `sarah`.


СОВЕТ

Программисты на C# говорят, что метод работает с *текущим* объектом. В первом вызове текущим объектом является `christa`, во втором — `sarah`.

Полное имя метода

Имеется тонкая, но важная проблема, связанная с описанием имен методов. Рассмотрим следующий фрагмент исходного текста:

```
public class Person
{
    public void Address()
    {
        Console.WriteLine("Hi");
    }
}
public class Letter
{
    string address;
    // Сохранение адреса
    public void Address(string newAddress)
    {
        address = newAddress;
    }
}
```

Любое обсуждение метода `Address()` после этого становится неоднозначным. Метод `Address()` класса `Person` не имеет ничего общего с методом `Address()` класса `Letter`. Если кто-то скажет, что в этом месте нужен вызов метода `Address()`, то какой именно метод `Address()` имеется в виду? Проблема не в самих методах, а в описании. Метода `Address()` как независимой самодостаточной сущности просто нет — есть методы `Person.Address()` и `Letter.Address()`. Путем добавления имени класса в начало имени метода явно указывается, какой именно метод имеется в виду.

Это описание имени метода очень похоже на описание имени человека. К примеру, в семье меня знают как Стефана. В семье больше нет Стефанов, и нет никакой неоднозначности, когда меня зовут по имени. Но на работе, где есть и другие Стефаны, чтобы избежать неоднозначности, следует добавлять к имени фамилию. Таким образом, `Address()` можно рассматривать как имя метода, а его класс — как фамилию.

Обращение к текущему объекту

Рассмотрим следующий метод `Student.SetName()`:

```
class Student
{
    // Информация об имени студента
    public string firstName;
    public string lastName;

    // SetName - сохранение информации об имени
    public void SetName(string fName, string lName)
    {
        firstName = fName;
        lastName = lName;
    }
}
public class Program
{
    public static void Main()
    {
        Student student1 = new Student();
        student1.SetName("Joseph", "Smith");
        Student student2 = new Student();
        student2.SetName("John", "Davis");
    }
}
```

Метод `Main()` использует метод `SetName()` для того, чтобы обновить поля объектов `student1` и `student2`. Но внутри метода `SetName()` нет ссылки ни на какой объект типа `Student`. Как уже было выяснено, метод работает "с текущим объектом". Но откуда он знает, какой именно объект — текущий? Ответ прост. Текущий объект передается при вызове метода как неявный аргумент; например, вызов

```
student1.SetName("Joseph", "Smith");
```

эквивалентен следующему:

```
Student.SetName(student1, "Joseph", "Smith");
// Это - эквивалентный вызов (однако он не будет
// корректно скомпилирован)
```

Я не хочу сказать, что вы можете вызвать метод `SetName()` двумя способами, я просто подчеркиваю, что эти два вызова семантически эквивалентны. Объект, являющийся текущим (скрытый первый аргумент), передается методу так же, как и другие аргументы. Оставьте эту задачу компилятору.

А что можно сказать о вызове одного метода из другого? Этот вопрос иллюстрируется следующим фрагментом исходного текста:

```
public class Student
{
    public string firstName;
    public string lastName;
    public void SetName(string firstName, string lastName)
    {
        SetFirstName(firstName);
        SetLastName(lastName);
    }
    public void SetFirstName(string name)
    {
        firstName = name;
    }
    public void SetLastName(string name)
    {
        lastName = name;
    }
}
```

В вызове `SetFirstName()` не видно никаких объектов. Дело в том, что при вызове одного метода объекта из другого в качестве неявного текущего объекта передается тот же объект, что и для вызывающего метода. Обращение к любому члену в методе объекта рассматривается как обращение к текущему объекту, так что метод сам знает, какому именно объекту он принадлежит.

Ключевое слово `this`

В отличие от других аргументов, текущий объект в список аргументов метода не попадает, а значит, программист не назначает ему никакого имени. Однако C# не оставляет этот объект безымянным и присваивает ему не слишком впечатляющее имя `this`, которое может пригодиться в ситуациях, когда вам нужно непосредственно обратиться к текущему объекту. Таким образом, можно переписать рассматривавшийся выше пример следующим образом:

```
public class Student
{
    public string firstName;
    public string lastName;
    public void SetName(string firstName, string lastName)
    {
        // Явная ссылка на "текущий объект" с применением
        // ключевого слова this
        this.SetFirstName(firstName);
        this.SetLastName(lastName);
    }
    public void SetFirstName(string name)
    {
        this.firstName = name;
    }
    public void SetLastName(string name)
```

```
    {
        this.lastName = name;
    }
}
```

Обратите внимание на явное добавление ключевого слова `this`. Добавление `this` к ссылкам на члены не привносит ничего нового, поскольку наличие `this` подразумевается и так. Однако, когда `Main()` делает показанный ниже вызов, `this` означает `student1` как в методе `SetName()`, так и в любом другом методе, который может быть вызван:

```
student1.SetName("John", "Smith");
```

ВНИМАНИЕ!

Ключевое слово C# `this` не может использоваться ни для какой иной цели, кроме описываемой.

Когда `this` используется явно

Обычно явно использовать `this` не требуется, так как компилятор достаточно разумен, чтобы разобраться в ситуации. Однако имеются две распространенные ситуации, когда это следует делать. Например, ключевое слово `this` может потребоваться при инициализации членов данных:

```
class Person
{
    public string name;
    public int id;
    public void Init(string name, int id)
    {
        this.name = name; // Имена аргументов те же,
                          // что и имена членов-данных
        this.id = id;
    }
}
```

Аргументы метода `Init()` носят имена `name` и `id`, которые совпадают с именами соответствующих членов-данных. Это повышает удобочитаемость, поскольку сразу видно, в какой переменной какой аргумент следует сохранить. Единственная проблема состоит в том, что имя `name` имеется как в списке аргументов, так и среди членов-данных. Такая ситуация оказывается слишком сложной для компилятора.

ЗАПОМНИ!

Добавление `this` проясняет ситуацию, четко определяя, что именно подразумевается под `name`. В `Init()` имя `name` означает аргумент метода, в то время как `this.name` — член объекта. Ключевое слово `this` также необходимо при сохранении текущего объекта для применения в дальнейшем или для использования в некотором другом методе. Рассмотрим следующую демонстрационную программу:

```
// ReferencingThisExplicitly - программа демонстрирует явное
// использование this
using System;

namespace ReferencingThisExplicitly
{
    public class Program
    {
        public static  void Main(string[] strings)
        {
            // Создание объекта студента
            Student student = new Student();
            student.Init("Stephen Davis", 1234);
            // Внесение курса в список
            Console.WriteLine("Внесение в список " +
                              "Stephen Davis " +
                              "курса Biology 101");
            student.Enroll("Biology 101");
            // Вывод прослушиваемого курса
            Console.WriteLine("Информация о студенте:");
            student.DisplayCourse();
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }

    // Student - класс, описывающий студента
    public class Student
    {
        // Все студенты имеют имена и идентификаторы
        public string  _name;
        public int     _id;

        // Курс, прослушиваемый студентом
        CourseInstance _courseInstance;

        // Init - инициализация объекта
        public void Init(string name, int id)
        {
            this._name = name;
            this._id   = id;
            _courseInstance = null;
        }

        // Enroll - добавление в список
        public void Enroll(string sCourseID)
        {
            _courseInstance = new CourseInstance();
            _courseInstance.Init(this, sCourseID);
        }
```

```
        // Вывод имени студента и прослушиваемых курсов
        public void DisplayCourse()
        {
            Console.WriteLine(_name);
            _courseInstance.Display();
        }
    }

    // CourseInstance - объединение информации
    // о студенте и прослушиваемом курсе
    public class CourseInstance
    {
        public Student _student;
        public string  _courseID;

        // Init - связь студента и курса
        public void Init(Student student, string courseID)
        {
            this._student   = student;
            this._courseID = courseID;
        }

        // Display - вывод имени курса
        public void Display()
        {
            Console.WriteLine(_courseID);
        }
    }
}
```

Это довольно приземленная программа. В объекте `Student` имеются поля для имени и идентификатора студента и один экземпляр курса (да, студент не очень ретивый...). Метод `Main()` создает экземпляр `student`, после чего вызывает метод `Init()` для его инициализации. В этот момент ссылка `_courseInstance` равна `null`, поскольку студенту еще не назначен ни один курс.

Метод `Enroll()` назначает студенту курс путем инициализации ссылки `_courseInstance` новым объектом. Однако метод `CourseInstance.Init()` получает экземпляр класса `Student` в качестве первого аргумента. Какой экземпляр должен быть передан? Очевидно, что следует передать текущий объект класса `Student`, т.е. именно тот, ссылкой на который является `this`.

СОВЕТ

Некоторые программисты предпочитают четко различать члены-данные и прочие переменные путем добавления ведущего или завершающего символа подчеркивания, например `_name` или `name_`. Само собой, это не более чем соглашение.

Что делать при отсутствии this

Смешивать статические методы классов и нестатические методы объектов — идея не из лучших, тем не менее C# и здесь может прийти на помощь. Чтобы понять, в чем состоит суть проблемы, давайте рассмотрим следующий исходный текст:

```
using System;
// MixingStaticAndInstanceMethods - совмещение методов
// класса и методов объектов может привести к проблемам
namespace MixingStaticAndInstanceMethods {

    public class Student
    {
        public string _firstName;
        public string _lastName;

        // InitStudent - инициализация объекта student
        public void InitStudent(string firstName, string lastName) {
            _firstName = firstName;
            _lastName = lastName;
        }

        // OutputBanner - вывод начальной строки
        public static void OutputBanner() {
            Console.WriteLine("Никаких хитростей:");

            // Вот где проблема -
            // Console.WriteLine(? какой объект используется ?);

        }

        // OutputBannerAndName - вывод начальной строки
        public void OutputBannerAndName() {

            // Используется класс Student, но статическому
            // методу не передаются никакие объекты
            OutputBanner();

            // Явная передача объекта
            OutputName(this);
        }

        // OutputName -- вывод имени студента
        public static void OutputName(Student student) {
            // Здесь объект указан явно
            Console.WriteLine("Имя студента - {0}",
                              student.ToNameString());
        }

        // ToNameString -- получение имени студента
        public string ToNameString() {
```

```
            // Здесь текущий объект указан неявно; можно
            // использовать this:
            // return this._firstName + " " + this._lastName;
            return _firstName + " " + _lastName;
        }
    }

    public class Program
    {
        public static void Main(string[] args) {
            Student student = new Student();
            student.InitStudent("Madeleine", "Cather");

            // Вывод заголовка и имени статически
            Student.OutputBanner();
            Student.OutputName(student);
            Console.WriteLine();

            // Вывод заголовка и имени через объект
            student.OutputBannerAndName();

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

Следует начать с конца программы, с метода `Main()`, чтобы лучше рассмотреть имеющиеся проблемы. Программа начинается с создания объекта класса `Student` и инициализации его имени. Затем она собирается всего лишь вывести имя студента с небольшим заголовком.

Метод `Main()` вначале выводит заголовок и сообщение с использованием статических методов. Программа вызывает метод `OutputBanner()` для вывода строки заголовка и `OutputName()` для вывода имени студента. Метод `OutputBanner()` просто выводит строку на консоль. Метод `OutputName()` получает в качестве аргумента объект класса `Student`, так что он может получить и вывести имя студента.

Далее метод `Main()` использует для решения той же задачи метод объекта и вызывает `student.OutputBannerAndName()`.

Затем `Main()` использует метод экземпляра для вывода заголовка и сообщения путем вызова `student.OutputBannerAndName()`. `OutputBannerAndName()` сначала вызывает статический метод `OutputBanner()` (предполагается класс `Student`). Ни один объект не передается, потому что статический метод `OutputBanner()` в нем не нуждается. Затем `OutputBannerAndName()` вызывает метод `OutputName()`. `OutputName()` также является статическим методом, но

он принимает в качестве аргумента объект `Student`. `OutputBannerAndName()` передает в качестве этого аргумента `this`.

Более интересная ситуация возникает при вызове `ToNameString()` из `OutputName()`. Метод `OutputName()` объявлен как `static` и, таким образом, не имеет `this`. У него есть явный объект класса `Student`, который он и использует для осуществления вызова.

Метод `OutputBanner()`, вероятно, также хотел бы вызвать `ToNameString()`, однако у него нет объекта `Student`. У него нет ссылки `this`, потому что это статический метод, и ему не передается объект. Обратите внимание на полужирную строку в коде: статический метод не может вызвать метод экземпляра.


ЗАПОМНИ!

Статический метод не может вызывать нестатические методы без явного указания объекта. Нет объекта — нет и вызова. В общем случае статический метод не может обратиться ни к одному нестатическому элементу класса. Однако нестатические методы могут обращаться как к статическим, так и к нестатическим членам класса — данным и методам.

Использование локальных функций

Несмотря на то что методы делают код меньше размером и проще в работе, иногда метод может оказаться слишком громоздким. Применение локальных функций позволяет объявить функцию в границах метода, чтобы способствовать дальнейшей инкапсуляции. Этот подход используется, когда в методе нужно выполнить несколько раз одну и ту же задачу, но никакой другой метод эту конкретную задачу больше не выполняет. Вот простой пример локальной функции:

```
static void Main(string[] args)
{
    // Создание локальной функции
    int Sum(int x, int y)
    {
        return x + y;
    }

    // Использование локальной функции для вывода некоторых сумм
    Console.WriteLine(Sum(1, 2));
    Console.WriteLine(Sum(5, 6));

    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}
```

Метод `Sum()` относительно прост, но его цель — демонстрация работы локальной функции. Функция инкапсулирует некоторый код, который использует только `Main()`. Поскольку `Main()` выполняет данную задачу более одного раза, использование `Sum()` позволяет сделать код более удобочитаемым, более понятным и более легким в обслуживании.

СОВЕТ

Локальные функции обладают всеми функциональными возможностями любого метода, за исключением того, что вы не можете объявлять их как статические. Локальная функция имеет доступ ко всем переменным, находящимся внутри метода, поэтому вы можете получить из `Sum()` доступ к переменным в `Main()`.

Глава 15

Класс: каждый сам за себя

В ЭТОЙ ГЛАВЕ...

- Защита класса
- Самостоятельная инициализация объекта
- Определение нескольких конструкторов
- Конструирование статических членов и членов класса
- Работа с членами с кодом

Класс должен сам отвечать за свои действия. Так же как микроволновая печь не должна вспыхнуть, объятая пламенем, из-за неверного нажатия кнопки, так и класс не должен скончаться (или прикончить программу) при предоставлении некорректных данных.

Чтобы нести ответственность за свои действия, класс должен убедиться в корректности своего начального состояния и в дальнейшем управлять им так, чтобы оно всегда оставалось корректным. C# предоставляет для этого все необходимое.

Ограничение доступа к членам класса

Простые классы определяют все свои члены как `public`. Рассмотрим программу `BankAccount`, которая поддерживает член-данные `balance` для хранения информации о балансе каждого счета. Сделав этот член `public`, вы допускаете любого в святая святых банка, позволяя каждому самому указывать сумму на счету.

Я не знаю ничего о вашем банке, но мой банк и близко не настолько открыт и всегда строго следит за моим счетом, самостоятельно регистрируя каждое снятие денег со счета и вклад на счет. В конце концов, это позволяет уберечься от всяких недоразумений, если вас вдруг подведет память.


ВНИМАНИЕ!

Вы можете решить, что достаточно лишь определить правило, согласно которому никакие другие классы не должны обращаться к члену `balance` непосредственно. Увы, теоретически это, может быть, и так, но на практике такой подход никогда не работает. Да, программисты начинают работу, преисполненные благими намерениями, которые вскоре непонятно куда исчезают под давлением сроков сдачи проекта...

Пример программы с использованием открытых членов

В приведенной демонстрационной программе класс `BankAccount` объявляет все методы как `public`, в то же время члены-данные `_accountNumber` и `_balance` сделаны `private`. Эта демонстрационная программа некорректна и не будет компилироваться, так как создана исключительно в дидактических целях.

```
// BankAccount - создание банковского счета с использованием
// переменной типа double для хранения баланса счета (она
// объявлена как private, чтобы скрыть баланс от внешнего
// мира)
// Примечание: пока в программу не будут внесены
// исправления, она не будет компилироваться, так как
// метод Main() обращается к private-члену класса
// BankAccount.
using System;

namespace BankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("В текущем состоянии эта " +
                              "программа не компилируется.");
```

```
        // Открытие банковского счета
        Console.WriteLine("Создание объекта " +
                          "банковского счета");
        BankAccount ba = new BankAccount();
        ba.InitBankAccount();
        // Обращение к балансу при помощи метода Deposit()
        // вполне корректно; Deposit() имеет право доступа ко
        // всем членам-данным
        ba.Deposit(10);
        // Непосредственное обращение к члену-данным вызывает
        // ошибку компиляции
        Console.WriteLine("Здесь вы получите " +
                          "ошибку компиляции");
        ba._balance += 10;
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
}

// BankAccount - определение класса, представляющего
// простейший банковский счет
public class BankAccount
{
    private static int _nextAccountNumber = 1000;
    private int _accountNumber;

    // Хранение баланса в виде одной переменной типа double
    private double _balance;

    // Init - инициализация банковского счета с нулевым
    // балансом и с использованием очередного глобального
    // номера
    public void InitBankAccount()
    {
        _accountNumber = ++_nextAccountNumber;
        _balance = 0.0;
    }

    // GetBalance - получение текущего баланса
    public double GetBalance()
    {
        return _balance;
    }

    // Номер счета
    public int GetAccountNumber()
    {
        return _accountNumber;
    }
    public void SetAccountNumber(int accountNumber)
    {
        this._accountNumber = accountNumber;
    }
```

```
        // Deposit - позволен любой положительный вклад
        public void Deposit(double amount)
        {
            if (amount > 0.0)
            {
                _balance += amount;
            }
        }

        // Withdraw - вы можете снять со счета любую сумму, не
        // превышающую баланс; метод возвращает реально снятую
        // сумму
        public double Withdraw(double withdrawal)
        {
            if (_balance <= withdrawal)
            {
                withdrawal = _balance;
            }

            _balance -= withdrawal;
            return withdrawal;
        }

        // GetString - возвращает информацию о состоянии счета в
        // виде строки
        public string GetString()
        {
            string s = String.Format("#{0} = {1:C}",
                                     GetAccountNumber(),
                                     GetBalance());
            return s;
        }
    }
}
```

Класс `BankAccount` предоставляет метод `InitBankAccount()` для инициализации членов класса, метод `Deposit()` — для обработки вкладов на счет и метод `Withdraw()` — для снятия денег со счета. Методы `Deposit()` и `Withdraw()` даже обеспечивают выполнение некоторых рудиментарных правил — "нельзя вкладывать отрицательные суммы" и "нельзя снимать больше, чем есть на счету". Однако в открытой системе, где член-данные `_balance` доступен для внешних методов (под *внешними* подразумеваются методы "в пределах той же программы, но внешние по отношению к классу"), эти правила могут быть нарушены кем угодно. Особенно существенная проблема может возникнуть при разработке больших проектов группами программистов. Это может стать проблемой и для одного человека, поскольку ему свойственно ошибаться.


ЗАПОМНИ!

Хорошо спроектированный код с правилами, выполнение которых проверяет компилятор, значительно снижает количество источников возможных ошибок. Перед тем как идти дальше, обратите внимание

на то, что приведенная демонстрационная программа не будет компилироваться — при такой попытке вы получите сообщение о том, что обращение к члену `DoubleBankAccount.BankAccount._balance` невозможно:

```
'BankAccount.BankAccount._balance' is inaccessible
        due to its protection level.
```

Трудно сказать, зачем компилятор заставили выводить такие скучные сообщения вместо короткого "не лезь к `private`", но суть именно в этом. Выражение `ba._balance += 10;` оказывается некорректным именно по этой причине — в силу объявления `_balance` как `private` этот член недоступен методу `Main()`, расположенному вне класса `BankAccount`. Замена данного выражения выражением `ba.Deposit(10)` решает возникшую проблему — метод `BankAccount.Deposit()` объявлен как `public`, а потому доступен для метода `Main()`.

ЗАПОМНИ!

Тип доступа по умолчанию — `private`, так что если вы забыли или сознательно пропустили модификатор для некоторого члена, это аналогично тому, как если бы вы описали его как `private`. Однако настоятельно рекомендуется всегда использовать это ключевое слово явно во избежание любых недоразумений. Хороший программист всегда явно указывает свои намерения, что является еще одним методом снижения количества возможных ошибок.

Прочие уровни безопасности

ВНИМАНИЕ!

В этом разделе используются определенные знания о наследовании и пространствах имен, которые будут рассмотрены в более поздних главах книги (глава 16, "Наследование", и 20, "Пространства имен и библиотеки"). Вы можете пропустить этот раздел и вернуться к нему позже, получив необходимые знания. Язык C# предоставляет следующие уровни безопасности.

- Члены, объявленные как `public`, доступны любому классу программы.
- Члены, объявленные как `private`, доступны только из текущего класса.
- Члены, объявленные как `protected`, доступны только из текущего класса и всех его подклассов.
- Члены, объявленные как `internal`, доступны для любого класса в том же модуле программы.

Модулем (module), или *сборкой* (assembly), в C# называется отдельно компилируемая часть кода, представляющая собой выполнимую

.EXE-программу либо библиотеку .DLL. Одно пространство имен может распространяться на несколько модулей. (В главе 20, "Пространства имен и библиотеки", рассматриваются сборки и пространства имен C# и обсуждаются уровни доступа, отличные от public и private.)

» Члены, объявленные как internal protected, доступны для текущего класса и всех его подклассов, а также классов в том же модуле программы.

Сокрытие членов путем объявления их как private обеспечивает максимальную степень безопасности. Однако зачастую такая высокая степень и не нужна. В конце концов, члены подклассов и так зависят от членов базового класса, так что ключевое слово protected предоставляет достаточно удобный уровень безопасности.

Зачем нужно управление доступом

Объявление внутренних членов класса как public — не лучшая мысль как минимум по следующим причинам.

» **Объявляя члены-данные public, вы не в состоянии просто определить, когда и как они модифицируются**. Зачем беспокоиться и создавать методы Deposit() и Withdraw() с проверками корректности? И вообще, зачем создавать любые методы, ведь любой метод любого класса может модифицировать данные счета в любой момент. Но если другой метод может обращаться к этим данным, то он практически обязательно это сделает.

Ваша программа BankAccount может проработать длительное время, прежде чем вы заметите, что баланс одного из счетов отрицателен. Метод Withdraw() призван оградить от подобной ситуации, но в описанном случае непосредственный доступ к балансу, минуя метод Withdraw(), имеют и другие методы. Вычислить, какие именно методы и при каких условиях поступают так некорректно, — задача не из легких.

» **Доступ ко всем членам-данным класса делает его интерфейс слишком сложным**. Как программист, использующий класс BankAccount, вы не хотите знать о том, что делается внутри него. Вам достаточно знаний о том, как положить деньги на счет и снять их с него.

» **Доступ ко всем членам-данным класса приводит к "растеканию" правил класса**. Например, класс BankAccount не позволяет балансу стать отрицательным ни при каких условиях. Это —

бизнес-правило, которое должно быть локализовано в методе `Withdraw()`. В противном случае вам придется добавлять соответствующую проверку в весь код, в котором осуществляется изменение баланса.

Что произойдет, если банк решит изменить правила и часть клиентов с хорошей кредитной историей получит право на небольшой отрицательный баланс в течение короткого времени? Вам придется долго рыскать по всей программе и вносить изменения во все места, где выполняется непосредственное обращение к балансу.


СОВЕТ

Не делайте классы и методы более доступными, чем это необходимо. Это не параноидальная боязнь хакеров — это просто поможет вам снизить количество ошибок в коде. По возможности используйте модификатор `private`, а затем при необходимости поднимайте его до `protected`, `internal`, `internal protected` или `public`.

Методы доступа

Если вы более внимательно посмотрите на класс `BankAccount`, то увидите несколько других методов. Один из них, `GetString()`, возвращает строковую версию счета для вывода ее на экран посредством вызова `Console.WriteLine()`. Дело в том, что вывод содержимого объекта `BankAccount` может быть затруднен, если это содержимое недоступно. К тому же, следуя принципу "отдайте кесарю кесарево", класс должен иметь право сам решать, как он будет представлен при выводе.

Кроме того, имеется два метода для *получения значения*, `GetBalance()` и `GetAccountNumber()`, и метод *установки значения* — `SetAccountNumber()`. Вы можете удивиться: зачем так волноваться из-за того, что член `_balance` будет объявлен как `private`, и при этом предоставлять метод `GetBalance()`? На самом деле для этого имеются достаточно веские основания.

- **`GetBalance()` не дает возможности изменять член `_balance` — он только возвращает его значение.** Тем самым значение баланса делается доступным только для чтения. Используя аналогию с настоящим банком, вы можете просмотреть состояние своего счета в любой момент, но не можете снять с него деньги иначе, чем с применением процедур, предусмотренных для этого банком.
- **Метод `GetBalance()` скрывает внутренний формат класса от внешних методов.** Метод `GetBalance()` может в процессе работы выполнять некоторые вычисления, обращаться к базе данных банка — словом, выполнять какие-то действия, чтобы получить состояние счета. Внешние методы ничего об этом не знают и не должны знать. Продолжая аналогию, вы интересуетесь состоянием счета, но не знаете, как, где и в каком именно виде хранятся ваши деньги.

И наконец, метод `GetBalance()` предоставляет механизм для внесения внутренних изменений в класс `BankAccount`, абсолютно не затрагивая при этом его пользователей. Если от национального банка придет распоряжение хранить деньги как-то иначе, это никак не должно сказаться на вашем способе обращения со счетом.

Пример управления доступом

Приведенная далее демонстрационная программа `DoubleBankAccount` указывает потенциальные изъяны программы `BankAccount`. В листинге показан только метод `Main()` — единственная претерпевшая изменения часть программы:

```
// DoubleBankAccount - создание банковского счета с
// использованием переменной типа double для хранения
// баланса счета (она объявлена как private, чтобы скрыть
// баланс от внешнего мира)
using System;
namespace DoubleBankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Открытие банковского счета
            Console.WriteLine("Создание объекта " +
                              "банковского счета");
            BankAccount ba = new BankAccount();
            ba.InitBankAccount();

            // Вклад на счет
            double deposit = 123.454;
            Console.WriteLine("Depositing {0:C}", deposit);
            ba.Deposit(deposit);

            // Баланс счета
            Console.WriteLine("Счет = {0}", ba.GetString());

            // Вот где возникает проблема
            double fractionalAddition = 0.002;
            Console.WriteLine("Adding {0:C}", fractionalAddition);
            ba.Deposit(fractionalAddition);

            // Результат
            Console.WriteLine("В результате счет = {0}",
                              ba.GetString());

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

Метод `Main()` создает банковский счет и вносит на него сумму 123,454, т.е. сумму с дробным количеством копеек. Затем метод `Main()` вносит на счет еще одну долю копейки и выводит баланс счета. Вывод программы выглядит следующим образом:

```
Создание объекта банковского счета
Вклад $123.45
Счет = #1001 = $123.45
Вклад $0.00
В результате счет = #1001 = $123.46
Нажмите <Enter> для завершения программы...
```

Пользователь начинает жаловаться на некорректные расчеты. Похоже, в программе имеется ошибка.

Проблема, конечно, в том, что `123.454` выводится как `123.45`. Чтобы избежать проблем, банк принимает решение округлять вклады и снятия до ближайшей копейки. Простейший путь осуществить это — конвертировать счета в `decimal` и использовать метод `Decimal.Round()`, как это сделано в демонстрационной программе `DecimalBankAccount`.

```
// DecimalBankAccount - создание банковского счета с
// использованием переменной типа decimal для хранения
// баланса счета
using System;

namespace DecimalBankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Открытие банковского счета
            Console.WriteLine("Создание объекта " +
                              "банковского счета");
            BankAccount ba = new BankAccount();
            ba.InitBankAccount();

            // Вклад на счет
            double deposit = 123.454;
            Console.WriteLine("Вклад {0:C}", deposit);
            ba.Deposit(deposit);

            // Баланс счета
            Console.WriteLine("Счет = {0}", ba.GetString());

            // Добавляем очень малую величину
            double fractionalAddition = 0.002;
            Console.WriteLine("Вклад {0:C}", fractionalAddition);
            ba.Deposit(fractionalAddition);
```

```
        // Результат
        Console.WriteLine("В результате счет = {0}",
                          ba.GetString());

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
}

// BankAccount - определение класса, представляющего
// простейший банковский счет
public class BankAccount
{
    private static int _nextAccountNumber = 1000;
    private int _accountNumber;

    // Хранение баланса в виде одной переменной типа decimal
    private decimal _balance;

    // Init - инициализация банковского счета с нулевым
    // балансом и использованием очередного глобального
    // номера
    public void InitBankAccount()
    {
        _accountNumber = ++_nextAccountNumber;
        _balance = 0;
    }

    // GetBalance - получение текущего баланса
    public double GetBalance()
    {
        return (double)_balance;
    }

    // AccountNumber
    public int GetAccountNumber()
    {
        return _accountNumber;
    }
    public void SetAccountNumber(int accountNumber)
    {
        this._accountNumber = accountNumber;
    }

    // Deposit - позволен любой положительный вклад
    public void Deposit(double amount)
    {
        if (amount > 0.0)
        {
            // Округление до ближайшей копейки перед
            // внесением вклада
```

```
            decimal temp = (decimal)amount;
            temp = Decimal.Round(temp, 2);
            _balance += temp;
        }
    }

    // Withdraw - вы можете снять со счета любую сумму, не
    // превышающую баланс; метод возвращает реально снятую
    // сумму
    public double Withdraw(double withdrawal)
    {
        // Преобразуем в тип decimal и работаем с ним.
        decimal decWithdrawal = (decimal)withdrawal;

        if (_balance <= decWithdrawal)
        {
            decWithdrawal = _balance;
        }

        _balance -= decWithdrawal;
        return (double)decWithdrawal; // Возврат double
    }

    // GetString - возвращает информацию
    // о состоянии счета в виде строки
    public string GetString()
    {
        string s = String.Format("#{0} = {1:C}",
                                 GetAccountNumber(),
                                 GetBalance());
        return s;
    }
}
}
```

Внутреннее представление поменялось на использование значений типа `decimal`, который в любом случае более подходит для работы с банковским счетом, чем тип `double`. Метод `Deposit()` теперь применяет метод `Decimal.Round()` для округления вкладываемой суммы до ближайшей копейки. Вывод программы оказывается таким, как и ожидалось:

```
Создание объекта банковского счета
Вклад $123.45
Счет = #1001 = $123.45
Вклад $0.00
В результате счет = #1001 = $123.45
Нажмите <Enter> для завершения программы...
```

Выводы

Вы можете сказать, что нужно было с самого начала писать программу `BankAccount` с использованием `decimal`, и, пожалуй, с вами можно согласиться. Но дело не в этом. Могут быть разные приложения и ситуации. Главное, что класс `BankAccount` оказался в состоянии решить проблему так, что не пришлось вносить никаких изменений в использующую его программу (обратите внимание на то, что открытый интерфейс класса не изменился: методы `Balance()` и `Withdraw()` так и возвращают значения типа `double`, а `Deposit()` и `Withdraw()` принимают параметр типа `double`).

В данном случае единственным методом, на который потенциально влияло изменение при непосредственном обращении к балансу, является метод `Main()`, но в реальной программе могут существовать десятки таких методов, и они могут оказаться в не меньшем количестве модулей. В данном случае ни один из этих методов не должен изменяться, потому что исправление находится в пределах класса `BankAccount`, *открытый интерфейс* которого (его открытые методы) не изменился. Если бы методы обращались ко внутренним членам класса непосредственно, это было бы решительно невозможно.


ВНИМАНИЕ!

Внесение внутренних изменений в класс требует определенного тестирования использующего класс кода, несмотря на то, что в него не вносятся никакие модификации.

Определение свойств класса

Методы `Get`*X*`()` и `Set`*X*`()`, продемонстрированные в программе `BankAccount`, называются *методами доступа* (access methods). Хотя их использование теоретически является хорошей привычкой, на практике это зачастую приводит к грустным результатам. Судите сами — чтобы увеличить член `_accountNumber` на `1`, требуется писать следующий код:

```
SetAccountNumber(GetAccountNumber() + 1);
```

C# имеет конструкцию, называемую *свойством* и делающую использование методов доступа существенно более простым. Приведенный далее фрагмент кода определяет свойство `AccountNumber` доступным для чтения и записи:

```
public int AccountNumber        // Скобки не нужны
{
    get{return accountNumber;}  // Фигурные скобки и точка с запятой
    set{accountNumber = value;} // Здесь 'value' - ключевое слово
}
```

Раздел `get` реализуется при чтении свойства, а `set` — при записи. В приведенном далее фрагменте исходного текста свойство `Balance` является свойством только для чтения, так как здесь определен только раздел `get`:

```
public double Balance
{
    get
    {
        return (double)balance;
    }
}
```

Использование свойств выглядит следующим образом:

```
BankAccount ba = new BankAccount();
// Записываем свойство AccountNumber

ba.AccountNumber = 1001;
// Считываем оба свойства
Console.WriteLine("#{0} = {1:C}",ba.AccountNumber, ba.Balance);
```

Свойства `AccountNumber` и `Balance` очень похожи на открытые члены-данные как внешне, так и в использовании. Однако свойства позволяют классу защитить свои внутренние члены (так, член `_balance` остается при этом `private`). Обратите внимание, что `Balance` выполняет приведение типа — точно так же может производиться любое количество вычислений. Свойства вовсе не обязательно должны представлять собой одну строку кода и могут выполнять различные действия наподобие проверки входных данных.

СОВЕТ

По соглашению имена свойств начинаются с прописной буквы. Обратите также внимание, что свойства не имеют скобок: следует писать просто `Balance`, а не `Balance()`.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Свойства совсем не обязательно неэффективны. Компилятор C# может оптимизировать простой метод доступа так, что он будет генерировать не больше машинных команд, чем непосредственное обращение к члену. Это важно не только для прикладных программ, но и для самого C#. Библиотека C# широко использует свойства, и то же самое должны делать и вы, даже для обращения к членам-данным класса из методов этого же класса.

Статические свойства

Статические члены-данные (класса) могут быть доступны через статические свойства, как показано в следующем простейшем примере:

```
public class BankAccount
{
    private static int nextAccountNumber = 1000;
    public static int NextAccountNumber
    {
        get{return nextAccountNumber;}
    }
    // ...
}
```

Свойство `NextAccountNumber` доступно посредством указания имени его класса, так как оно не является свойством конкретного объекта (оно объявлено как `static`).

```
// Считываем свойство NextAccountNumber
int value = BankAccount.NextAccountNumber;
```

(В этом примере `value` находится вне контекста свойства, а потому не рассматривается как зарезервированное слово.)

Побочные действия свойств

Операция `get` может применяться не только для простого получения значения, связанного со свойством. Взгляните на следующий код:

```
public static int AccountNumber
{
    // Получение значения переменной и увеличение ее значения,
    // чтобы в следующий раз получить уже новое ее значение
    get{return ++_nextAccountNumber;}
}
```

Данное свойство увеличивает статический член класса перед тем, как вернуть результат. Однако это не слишком хорошая идея, ведь пользователь ничего не знает о такой особенности и не подозревает, что происходит что-то помимо чтения значения. Увеличение переменной в данном случае представляет собой *побочное действие*.

ЗАПОМНИ!

Подобно методам доступа, которые они имитируют, свойства не должны изменять состояния класса иначе чем через установку значения соответствующего члена данных. В общем случае и свойства, и методы должны избегать побочных действий, так как это может привести к трудноуловимым ошибкам. Изменяйте класс настолько явно и непосредственно, насколько это возможно.

Дайте компилятору написать свойства для вас

Большинство свойств, описанных в предыдущем разделе, представляют собой простые подпрограммы, писать которые очень просто... и утомительно:

```
private string _name;     // Член, соответствующий свойству
public string Name { get { return _name; } set { _name = value; }}
```

Поскольку код везде оказывается одним и тем же, было решено позволить компилятору C# 3.0 делать эту работу вместо вас. Вот все, что вы должны написать:

```
public string Name { get; set; }
```

Это эквивалентно

```
private string <somename>;     // Что такое <somename>?
                               // неизвестно и неважно.
public string Name { get { return <somename>; }
                     set { <somename> = value; } }
```

Компилятор создает некий загадочный член-данные, который во всем приведенном коде оказывается безымянным. Такой стиль заставляет использовать свойства даже внутри других членов того же класса просто потому, что все, что вам известно, — это имя свойства. По этой причине вы должны иметь оба свойства — и `get`, и `set`. Инициализировать их можно при помощи следующего синтаксиса:

```
public int AnInt {get; set;} // Компилятор создает
                             // закрытую переменную
. . .
AnInt = 2;                   // Инициализация созданной компилятором
                             // переменной при помощи свойства.
```

Методы и уровни доступа

Методы доступа не обязательно должны быть объявлены как `public`. Вы можете объявлять их на любом уровне доступа, включая `private`, если метод доступа предназначен для использования исключительно внутри собственного класса.

Можно даже отдельно изменять уровень доступа для частей `get` и `set`. Предположим, например, что вы не хотите давать возможность работы с методом `set` вне класса. В этом случае свойство можно записать таким образом:

```
internal string Name { get; private set; }
```

Конструирование объектов с помощью конструкторов


ЗАПОМНИ!

Управление доступом — это только половина проблемы. Рождение объекта — один из самых важных этапов в его жизни. Класс, конечно, может предоставить метод для инициализации вновь созданного объекта, но беда в том, что приложение может попросту забыть его вызвать. В таком случае члены-данные класса окажутся заполненными "мусором", и корректной работы от такого объекта ждать не придется. Язык C# решает эту проблему путем вызова инициализирующего метода автоматически, например:

```
MyObject mo = new MyObject();
```

Эта инструкция не только выделяет память для объекта, но и выполняет инициализацию его членов.


ЗАПОМНИ!

Не путайте термины *класс* и *объект*. `Cat` — это класс, но экземпляр класса `Cat` по имени `Striper` — это объект класса `Cat`.

Конструкторы, предоставляемые C#

Язык C# хорошо умеет отслеживать инициализацию переменных и не позволяет использовать неинициализированные переменные. Например, представленный далее код приведет к генерации ошибки времени компиляции:

```
public static void Main(string[] args)
{
    int n;
    double d;
    double calculatedValue = n + d;
}
```

Язык C# отслеживает тот факт, что ни `n`, ни `d` не имеют присвоенного значения и не могут использоваться в выражении. Компиляция этой микропрограммы приводит к генерации следующих сообщений об ошибках:

```
Use of unassigned local variable 'n'
Use of unassigned local variable 'd'
```

C# предоставляет конструктор по умолчанию, который инициализирует члены данных объекта:

- числа — нулями;
- логические переменные — значениями `false`;
- ссылки на объекты — значениями `null`.

Рассмотрим следующую простую демонстрационную программу:

```
using System;
namespace Test
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Сначала создаем объект
            MyObject localObject = new MyObject();
            Console.WriteLine("localObject.n = {0}",
                              localObject.n);
            if (localObject.nextObject == null)
            {
                Console.WriteLine("localObject.nextObject = null");
            }
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
    public class MyObject
    {
        internal int n;
        internal MyObject nextObject;
    }
}
```

Эта программа определяет класс `MyObject`, который содержит переменную `n` типа `int` и ссылку на объект `nextObject`, позволяющую создавать связанные списки объектов. Метод `Main()` создает объект класса `MyObject` и выводит начальное содержимое его членов. Вывод этой программы имеет следующий вид:

```
localObject.n = 0
localObject.nextObject = null
Нажмите <Enter> для завершения программы...
```

Язык C# при создании объекта выполняет небольшой код по инициализации объекта и его членов. Если бы не этот код, члены-данные `localObject.n` и `localObject.nextObject` содержали бы какие-то случайные значения, попросту говоря — "мусор".

ЗАПОМНИ!

Код, инициализирующий значения при создании, называется *конструктором по умолчанию*. Он "конструирует" класс в смысле инициализации его членов. Таким образом, C# гарантирует, что объект начинает жизнь в известном состоянии — полностью обнуленным. *Это относится только к данным-членам класса, но не к локальным переменным метода.*

Замена конструктора по умолчанию

Хотя компилятор автоматически инициализирует все переменные экземпляров соответствующими значениями, для многих классов (возможно, даже для большинства) значения по умолчанию не являются корректным состоянием. Рассмотрим класс `BankAccount`, о котором уже шла речь в этой главе.

```
public class BankAccount
{
    private int _accountNumber;
    private double _balance;
    // ... прочие члены
}
```

Хотя нулевое начальное значение баланса вполне корректно, нулевое значение номера счета, определенно, не является верным.

Поэтому в данный момент класс `BankAccount` включает метод `InitBankAccount()`, инициализирующий объект. Однако такой подход перекладывает слишком большую ответственность на прикладную программу, использующую данный класс. Если вдруг приложение забудет вызвать метод `InitBankAccount()`, то прочие методы банковского счета могут оказаться неработоспособными, хотя при этом и не будут содержать никаких ошибок.

ЗАПОМНИ!

Класс не должен полагаться на внешние методы наподобие метода `InitBankAccount()`, которые должны обеспечивать корректное состояние его объектов. Для решения данной проблемы класс предоставляет специальный метод, автоматически вызываемый C# при создании объекта, — *конструктор класса*. Конструктор мог бы именоваться как `Init()`, `Start()` или `Create()`, но C# требует, чтобы конструктор носил то же имя, что и имя самого класса, так что конструктор класса `BankAccount` имеет следующий вид:

```
public void Main(string[] args)
{
    BankAccount ba = new BankAccount(); // Вызов конструктора
}
```

```
public class BankAccount
{
    // Номера банковских счетов начинаются с 1000 и
    // назначаются последовательно в возрастающем порядке
    static int nextAccountNumber = 1000;
    // Для каждого счета поддерживаются его номер и баланс
    int accountNumber;
    double balance;
    // Конструктор BankAccount - обратите внимание на его имя
    public BankAccount() // Требуются круглые скобки, могут
                         // иметься аргументы, возвращаемый
                         // тип отсутствует
    {
        accountNumber = ++nextAccountNumber;
        balance = 0.0;
    }
    // . . . прочие члены . . .
}
```

Содержимое конструктора `BankAccount` то же, что и первоначального метода `InitBankAccount()`. Однако конструктор имеет некоторые особенности:

- всегда имеет то же имя, что и сам класс;
- может как принимать параметры, так и вызываться без них;
- не имеет возвращаемого типа, даже типа `void`;
- метод `Main()` не должен вызывать никаких дополнительных методов для инициализации объекта при его создании — не нужны никакие вызовы `Init()`.


ЗАПОМНИ!

Если вы создаете собственный конструктор, C# не создает конструктор по умолчанию автоматически. Ваш конструктор заменяет конструктор по умолчанию и становится единственным способом создания экземпляра класса.

Конструирование объектов

Теперь посмотрим на конструкторы в деле. Для этого рассмотрим программу `DemonstrateCustomConstructor`.

```
using System;
// DemonstrateCustomConstructor -- демонстрация работы
// конструкторов по умолчанию; создаем класс с конструктором
// и рассматриваем несколько сценариев.
namespace DemonstrateCustomConstructor
{
    // MyObject - создание класса с "многословным"
    // конструктором и внутренним объектом
    public class MyObject
    {
```

```
        // Этот член-данные является свойством класса
        private static MyOtherObject _staticObj =
                                            new MyOtherObject();

        // Этот член-данные является свойством каждого объекта
        private MyOtherObject _dynamicObj;

        // Конструктор (с обильным выводом на экран)
        public MyObject()
        {
            Console.WriteLine("Начало конструктора MyObject");
            Console.WriteLine("(Статические члены-данные " +
                              "конструируются до этого " +
                              "конструктора)");
            Console.WriteLine("Теперь динамически создаем " +
                              "нестатический член-данные:");
            _dynamicObj = new MyOtherObject();
            Console.WriteLine("MyObject constructor ending");
        }
    }

    // MyOtherObject - у этого класса тоже многословный
    // конструктор, но внутренние члены-данные отсутствуют
    public class MyOtherObject
    {
        public MyOtherObject()
        {
            Console.WriteLine("Конструирование MyOtherObject");
        }
    }
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Начало метода Main()");
            Console.WriteLine("Создание локального объекта " +
                              "MyObject в Main():");
            MyObject localObject = new MyObject();

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

Выполнение данной программы приводит к следующему выводу на экран:

```
Начало метода Main()
Создание локального объекта MyObject в Main():
Конструирование MyOtherObject
Начало конструктора MyObject
```

```
(Статические члены-данные конструируются до этого конструктора)
Теперь динамически создаем нестатический член-данные:
Конструирование MyOtherObject
Завершение конструктора MyObject
Нажмите <Enter> для завершения программы...
```

Вот реконструкция происходящего при запуске программы.

1. **Программа начинает работу, и метод `Main()` выводит начальное сообщение и сообщение о предстоящем создании локального объекта `MyObject`.**
2. **Метод `Main()` создает объект `localObject` типа `MyObject`.**
3. **`MyObject` содержит статический член `_staticObj` класса `MyOtherObject`.**

   Все статические члены-данные инициализируются до первого выполнения конструктора `MyObject()`. В этом случае C# присваивает переменной `_staticObj` ссылку на вновь созданный объект перед тем, как передать управление конструктору `MyObject`.
4. **Конструктор `MyObject` получает управление. Он выводит начальное сообщение и напоминает, что статический член уже сконструирован до того, как начал работу конструктор `MyObject()`.**
5. **После объявления о своих намерениях по динамическому созданию нестатического члена конструктор `MyObject` создает объект класса `MyOtherObject` с использованием оператора `new`, что сопровождается выводом второго сообщения о создании `MyOtherObject` на экран.**
6. **Управление возвращается конструктору `MyObject`, который, в свою очередь, возвращает управление методу `Main()`.**

Непосредственная инициализация объекта

Помимо инициализации членов-данных в конструкторе, C# позволяет инициализировать члены-данные непосредственно с использованием инициализаторов. Это означает, что класс `BankAccount` можно записать следующим образом:

```
public class BankAccount
{
    // Номера банковских счетов начинаются с 1000 и
    // назначаются последовательно в возрастающем порядке
    static int _nextAccountNumber = 1000;

  // Для каждого счета поддерживаются его номер и баланс
  int _accountNumber = ++_nextAccountNumber;
  double _balance = 0.0;

  // ... прочие члены ...
}
```

Вот в чем состоит работа инициализаторов. Как _accountNumber, так и _balance получают значения как часть объявления, эффект которого аналогичен использованию указанного кода в конструкторе.

Надо очень четко представлять себе картину происходящего. Вы можете решить, что это выражение присваивает значение 0.0 переменной _balance непосредственно. Но ведь _balance существует только как часть некоторого объекта. Таким образом, присваивание не выполняется до тех пор, пока не будет создан объект _BankAccount. Рассматриваемое присваивание осуществляется всякий раз при создании объекта.

Заметим, что статический член-данные _nextAccountNumber инициализируется при первом обращении к классу BankAccount (как вы убедились при выполнении демонстрационной программы в отладчике), т.е. при обращении к любому свойству или методу объекта, владеющему статическим членом, в том числе к конструктору.


ЗАПОМНИ!

Будучи инициализированным, статический член повторно не инициализируется, сколько бы объектов вы ни создавали. Этим он отличается от нестатических членов. Инициализаторы выполняются в порядке их появления в объявлении класса. Если C# встречает и инициализаторы, и конструктор, то инициализаторы выполняются раньше тела конструктора.

Конструирование с инициализаторами

Давайте в программе DemonstrateCustomConstructor перенесем вызов new MyOtherObject() из конструктора MyObject в объявление так, как показано в приведенном далее фрагменте исходного текста полужирным шрифтом, и изменим второй вызов WriteLine().

```
public class MyObject
{
    // Этот член является свойством класса
    private static MyOtherObject _staticObj = new MyOtherObject();

    // Этот член является свойством объекта
    private MyOtherObject _dynamicObj = new MyOtherObject();

    public MyObject()
    {
        Console.WriteLine("Начало конструктора MyObject");
        Console.WriteLine("(Статические члены " +
                          "инициализированы до конструктора)");
        // Ранее здесь создавался _dynamicObj
        Console.WriteLine("Завершение конструктора MyObject");
    }
}
```

Сравните вывод на экран такой модифицированной программы с выводом на экран исходной программы DemonstrateCustomConstructor:

```
Начало метода Main()
Создание локального объекта MyObject в Main():
Конструирование MyOtherObject
Конструирование MyOtherObject
Начало конструктора MyObject
(Статические члены инициализированы до конструктора)
Завершение конструктора MyObject
Нажмите <Enter> для завершения программы...
```

Инициализация объекта без конструктора

Предположим, у вас есть небольшой класс для представления студента:

```
public class Student
{
    public string Name { get; set; }
    public string Address { get; set; }
    public double GradePointAverage { get; set; }
}
```

Объект Student имеет три открытых свойства, Name, Address и GradePoint Average, которые содержат всю основную информацию о студенте. Обычно при создании нового объекта Student вы должны инициализировать его свойства Name, Address и GradePointAverage примерно таким образом:

```
Student randal = new Student();
randal.Name = "Randal Sphar";
randal.Address = "123 Elm Street, Truth or Consequences, NM 00000";
randal.GradePointAverage = 3.51;
```

Если класс Student имеет конструктор, можно поступить следующим образом:

```
Student randal = new Student("Randal Sphar",
        "123 Elm Street, Truth or Consequences, NM, 00000", 3.51);
```

Однако, увы, у класса Student нет другого конструктора, кроме конструктора по умолчанию, автоматически создаваемого C#, и не принимающего никаких аргументов.


ЗАПОМНИ!

В C# 3.0 и более поздних версиях можно упростить такую инициализацию при помощи кода, выглядящего подозрительно похожим на конструктор:

```
Student randal = new Student
    { Name = "Randal Sphar",
      Address = "123 Elm Street, Truth or Consequences, NM 00000",
      GradePointAverage = 3.51
    };
```

Чем отличаются эти два примера? Первый, использующий конструктор, содержит *круглые скобки*, в которые заключены две строки и одно число с плавающей точкой, разделенные запятыми. Во втором примере с применением нового синтаксиса инициализации вместо этого используются *фигурные скобки*, в которых содержатся три *присваивания*, разделенные запятыми. Этот синтаксис работает следующим образом:

```
new LatitudeLongitude
    { присваивание Latitude, присваивание Longitude };
```

Данный синтаксис инициализации объектов позволяет выполнять присваивание любому разрешающему присваивание свойству (*set*) объекта в блоке кода (в фигурных скобках). Этот блок предназначен для инициализации объекта. Заметим, что таким образом можно назначать значения только открытым свойствам, но не закрытым, а кроме того, в этом коде нельзя вызывать никакие методы объекта или выполнять какую-то иную работу.

Такой синтаксис весьма краток — одна инструкция вместо трех. Он упрощает создание инициализированных объектов, которые вы не можете инициализировать при помощи конструктора. Дает ли новый синтаксис инициализации что-либо, кроме удобства? Не многое, но удобство всегда находится в верху списка предпочтений практикующего программиста (так же, как и краткость). Кроме того, эта возможность очень важна при работе с анонимными классами.


СОВЕТ

Пользуйтесь этой возможностью свободно, так, как вам подсказывает ваша интуиция. Если вы хотите узнать о ней побольше, поищите в справочной системе *object initializer*.

Применение членов с кодом

Члены с кодом (expression-bodied members) впервые появились в C# 6.0 как средство, облегчающее определение методов и свойств. В C# 7.0 члены с кодом работают также с конструкторами, деструкторами, методами доступа к свойствам и событиям.

Создание методов с кодом

В приведенном примере показано, как можно было создавать методы до C# 6.0:

```
public int RectArea(Rectangle rect)
{
    return rect.Height * rect.Width;
}
```


ЗАПОМНИ!

При работе с членами с кодом можно уменьшить количество строк кода до одной:

```
public int RectArea(Rectangle rect) => rect.Height * rect.Width;
```

Хотя обе версии выполняют одно и то же действие, вторая версия намного короче и ее легче написать. Компромисс заключается в том, что вторая версия может быть сложнее для понимания.

Определение свойств с кодом

Свойства с кодом работают подобно методам: вы объявляете свойство с помощью единственной строки кода:

```
public int RectArea => _rect.Height * _rect.Width;
```

В этом примере предполагается, что у нас определен закрытый член `_rect` и что вы хотите получить значение, равное площади прямоугольника.

Определение конструкторов и деструкторов с кодом

В C# 7.0 можно использовать тот же подход для работы с конструктором. В более ранних версиях C# можно создавать конструктор следующим образом:

```
public EmpData()
{
    _name = "Harvey";
}
```

Здесь конструктор класса `EmpData` устанавливает значение закрытой переменной `_name` равным `"Harvey"`. C# 7.0 для этого достаточно одной строки:

```
public EmpData() => _name = "Harvey";
```

Деструкторы работают в основном так же, как и конструкторы. Вместо многих строк вы можете использовать только одну.

Определение методов доступа к свойствам с кодом

Методы доступа к свойствам также могут извлечь выгоду из членов с кодом. Вот типичный метод доступа в C# 6.0 с `get` и `set`:

```
private int _myVar;
public MyVar {
    get
    {
        return _myVar;
    }
    set
    {
        SetProperty(ref _myVar, value);
    }
}
```

А вот во что он превращается в C# 7.0 при использовании членов с кодом:

```
private int _myVar;
public MyVar
{
    get => _myVar;
    set => SetProperty(ref _myVar, value);
}
```

Определение методов доступа к событиям с кодом

Как и в случае доступа к свойствам, можно создавать средства доступа к событиям, используя член с кодом. Вот что могло быть использовано в C# 6.0:

```
private EventHandler _myEvent;
public event EventHandler MyEvent
{
    add
    {
        _myEvent += value;
    }
    remove
    {
        _myEvent -= value;
    }
}
```

И вот как выглядит тот же метод доступа к событию в C# 7.0:

```
private EventHandler _myEvent;
public event EventHandler MyEvent
{
    add    => _myEvent += value;
    remove => _myEvent -= value;
}
```

Глава 16

Наследование

В ЭТОЙ ГЛАВЕ...

- Определение одного класса через другой
- Разница между "является" и "содержит"
- Подстановка объекта одного класса вместо другого
- Построение статических членов и членов классов
- Включение конструкторов в иерархию наследования
- Вызов конструктора базового класса

Объектно-ориентированное программирование основано на четырех принципах: управления доступом (инкапсуляция), наследования других классов, возможности соответствующего отклика (полиморфизм) и возможности косвенного обращения одного объекта к другому (интерфейсы).

Наследование — распространенная концепция. Вы — человек. Вы наследуете ряд свойств класса `Human` (человек), таких как зависимость от воздуха, еды, умение разговаривать и т.п. Класс `Human` наследует потребность в воздухе, воде и еде от класса `Mammal` (млекопитающее), а тот, в свою очередь, — от класса `Animal` (животное).

Возможность такой передачи свойств очень важна. Она позволяет экономно описывать вещи и концепции. Например, на вопрос ребенка "Что такое утка?" можно ответить "Это птица, которая крякает". Независимо от того, что вы подумали о таком ответе, он содержит значительное количество информации.

Ребенок знает, что такое птица, а теперь он знает, что все то же самое можно сказать и об утке, а кроме того, у утки есть дополнительное свойство — "кряканье".

Объектно-ориентированные языки программирования выражают отношение наследования, позволяя одному классу наследовать другой, что, в свою очередь, дает возможность объектно-ориентированным языкам генерировать модели, более близкие к реальности, чем модели, генерируемые языками, объектно-ориентированное программирование не поддерживающими.

Наследование класса

В приведенной далее демонстрационной программе `InheritanceExample` класс `SubClass` наследован от класса `BaseClass`.

```
// InheritanceExample - простейшая демонстрация наследования
using System;

namespace InheritanceExample
{
  public class BaseClass
  {
    public int _dataMember;

    public void SomeMethod()
    {
      Console.WriteLine("SomeMethod()");
    }
  }

  public class SubClass : BaseClass
  {
    public void SomeOtherMethod()
    {
      Console.WriteLine("SomeOtherMethod()");
    }
  }

  public class Program
  {
    public static void Main(string[] args)
    {
      // Создание объекта базового класса
      Console.WriteLine("Работа с объектом " +
                        "базового класса:");
      BaseClass bc = new BaseClass();
      bc._dataMember = 1;
      bc.SomeMethod();
```

```
        // Создание объекта подкласса
        Console.WriteLine("Работа с объектом подкласса:");
        SubClass sc = new SubClass();
        sc._dataMember = 2;
        sc.SomeMethod();
        sc.SomeOtherMethod();

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
  }
}
```

Класс `BaseClass` определен как имеющий член-данные и простой метод `SomeMethod()`. Метод `Main()` создает объект `bc` базового класса `BaseClass` и работает с ним. Класс `SubClass` наследуется от класса `BaseClass` путем размещения имени класса `BaseClass` после двоеточия в определении класса.

```
public class SubClass : BaseClass
```

ПОТРЯСАЮЩЕЕ НАСЛЕДОВАНИЕ

Чтобы было проще разбираться в окружающем мире, люди составляют обширные системы. Тузик является частным случаем собаки, которая относится к собакообразным, входящим в состав млекопитающих, и т.д. Так легче познавать мир.

В объектно-ориентированных языках, таких как C#, говорится, что класс `Student` наследует класс `Person`. Кроме того, `Person` является *базовым классом* для класса `Student`. Наконец, можно сказать, что `Student` ЯВЛЯЕТСЯ `Person` (использование прописных букв — общепринятый метод отражения уникального типа связи). Эта терминология применяется в C++ и других объектно-ориентированных языках программирования.

Заметьте, что хотя `Student` и ЯВЛЯЕТСЯ `Person`, обратное не верно. `Person` НЕ ЯВЛЯЕТСЯ `Student` (такое выражение следует трактовать в общем смысле, поскольку конкретный человек, конечно же, может оказаться студентом). Существует много людей, являющихся членами класса `Person`, но не членами класса `Student`. Кроме того, класс `Student` обладает свойствами, которых нет у класса `Person`. Например, класс `Student` имеет средний балл, а `Person` — нет.

Свойство наследования транзитивно. Например, если определить новый класс `GraduateStudent` как подкласс класса `Student`, то он тоже будет наследником `Person`. Это значит, что будет выполняться следующее: если `GraduateStudent` ЯВЛЯЕТСЯ `Student` и `Student` ЯВЛЯЕТСЯ `Person`, то `GraduateStudent` ЯВЛЯЕТСЯ `Person`.

ЗАПОМНИ!

SubClass получает все члены класса BaseClass в качестве собственных, а также члены, которые могут быть в него добавлены. Метод Main() демонстрирует, что SubClass имеет член-данные _dataMember и член-метод SomeMethod(), унаследованные от класса BaseClass, а также новый метод SomeOtherMethod(), которого нет у базового класса. Вывод программы на экран выглядит так, как от него и ожидалось:

```
Работа с объектом базового класса:
SomeMethod()
Работа с объектом подкласса:
SomeMethod()
SomeOtherMethod()
Нажмите <Enter> для завершения программы...
```

Зачем нужно наследование

Наследование выполняет ряд важных функций. Вы можете решить, что главная из них — уменьшить количество ударов по клавишам в процессе ввода программы. И это тоже — вам не надо заново вводить все свойства Person при описании класса Student. Однако более важна возможность *повторного использования* (reuse). Нет нужды начинать каждый новый проект "с нуля", если можно воспользоваться готовыми программными компонентами.

Сравним разработку программного обеспечения с другими областями человеческой деятельности. Многие ли производители автомобилей начинают проектировать новую модель с разработки для этого новых шурупов, болтов и гаек? И даже если это так, то что вы скажете о разработке новых молотков, отверток и прочего инструментария? Конечно же, нет. По возможности при проектировании и сборке новой модели максимально используются детали и части старой — не только болты и гайки, но и крупные узлы, такие как компрессоры или даже двигатели.

Наследование позволяет настроить уже имеющиеся программные компоненты. Старые классы могут быть адаптированы для применения в новых программах без внесения в них кардинальных изменений. Существующий класс наследуется — с *расширением* его возможностей — новым подклассом, который содержит все необходимые добавления и изменения. Если базовый класс написан кем-то иным, у вас может просто не быть возможности вносить в него изменения, и наследование оказывается единственным способом его использования.

Данная возможность тесно связана с третьим преимуществом применения наследования. Представим ситуацию, когда вы наследуете базовый класс.

Позже выясняется, что в нем имеется ошибка, которую нужно исправить. Если вы модифицировали класс для его повторного использования, вы должны вручную внести изменения и протестировать каждое приложение в отдельности. При наследовании класса без внесения изменений вы в общем случае исправляете только базовый класс, не затрагивая сами приложения.

Однако главное преимущество наследования в том, что оно описывает реальный мир таким, каков он есть.

Более сложный пример наследования

Банк поддерживает несколько типов счетов. Один из них — депозитный счет — обладает всеми свойствами простого банковского счета плюс возможностью накопления процентов. Такое отношение на языке C# моделируется в приведенной далее демонстрационной программе `SimpleSavingsAccount`.

```
// SimpleSavingsAccount - реализация счета SavingsAccount
// как разновидности BankAccount; здесь не используются
// виртуальные методы (о них будет сказано в главе 13)
using System;
namespace SimpleSavingsAccount
{
    // BankAccount - модель банковского счета, который имеет
    // номер и хранит текущий баланс
    public class BankAccount          // Базовый класс
    {
        // Номера счетов начинаются с 1000 и образуют
        // возрастающую последовательность
        public static int _nextAccountNumber = 1000;
        // Номер счета и баланс для каждого объекта свои
        public int _accountNumber;
        public decimal _balance;
        // Init - инициализация счета очередным свободным
        // номером и конкретным начальным балансом
        // (по умолчанию — нуль)
        public void InitBankAccount()
        {
            InitBankAccount(0);
        }
        public void InitBankAccount(decimal initialBalance)
        {
            _accountNumber = ++_nextAccountNumber;
            _balance = initialBalance;
        }
        // Свойство Balance
        public decimal Balance
        {
            get { return _balance;}
        }
```

```
    // Deposit - позволен любой положительный вклад
    public void Deposit(decimal amount)
    {
        if (amount > 0)
        {
            _balance += amount;
        }
    }
    // Withdraw - можно снять не более того, что имеется на
    // счету; метод возвращает снятую сумму
    public decimal Withdraw(decimal withdrawal)
    {
        if (Balance <= withdrawal) // используется свойство
        {
            // Balance
            withdrawal = Balance;
        }

        _balance -= withdrawal;
        return withdrawal;
    }
    // ToString - строка с информацией о состоянии счета
    public string ToBankAccountString()
    {
        return String.Format("{0} - {1:C}",
                             _accountNumber, Balance);
    }
}
// SavingsAccount - банковский счет с накоплением
// процентов
public class SavingsAccount : BankAccount // Подкласс
{
    public decimal _interestRate;
    // InitSavingsAccount - использует процентную ставку,
    // выражаемую числом от 0 до 100
    public void InitSavingsAccount(decimal interestRate)
    {
        InitSavingsAccount(0, interestRate);
    }
    public void InitSavingsAccount(decimal initial,
                                   decimal interestRate)
    {
        InitBankAccount(initial);
        _interestRate = interestRate / 100;
    }
    // AccumulateInterest - вызывается однократно в конце
    // периода начисления процентов
    public void AccumulateInterest()
    {
        _balance = Balance + (decimal)(Balance * _interestRate);
    }
    // ToString - строка с информацией о состоянии счета
    public string ToSavingsAccountString()
```

```
        {
            return String.Format("{0} ({1}%)",
                                 ToBankAccountString(),
                                 _interestRate * 100);
        }
    }
    public class Program
    {
        public static void Main(string[] args)
        {
            // Создание банковского счета и вывод на экран
            BankAccount ba = new BankAccount();
            ba.InitBankAccount(100M); // Суффикс M говорит о
            ba.Deposit(100M);         // типе decimal
            Console.WriteLine("Счет {0}",
                              ba.ToBankAccountString());
            // Теперь - депозитный счет
            SavingsAccount sa = new SavingsAccount();
            sa.InitSavingsAccount(100M, 12.5M);
            sa.AccumulateInterest();
            Console.WriteLine("Счет {0}",
                              sa.ToSavingsAccountString());
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

Класс `BankAccount` ничем не отличается от того, каким он был в других главах книги. Он начинается с перегруженного метода инициализации `InitBankAccount()`: одного — для произвольного начального значения баланса, другого — для нулевого баланса. Обратите внимание на то, что здесь не использованы конструкторы. Окончательная версия `BankAccount` в этой главе решает проблему конструктора, но пока что в иллюстративных целях используется более простая версия.

Свойство `Balance` позволяет другим читать значение баланса, запрещая при этом его изменять. Метод `Deposit` принимает любой положительный вклад, а метод `Withdraw()` предоставляет возможность снять со счета любую сумму (в пределах наличного баланса). Метод `ToBankAccountString()` создает строку с описанием состояния счета.

Класс `SavingsAccount` наследует все, что можно, от класса `BankAccount`. К этому он добавляет процентную ставку и возможность накопления процентов.

Метод `Main()` делает минимально возможную работу: создает счет `BankAccount`, выводит информацию о нем, создает счет `SavingsAccount`, один

раз начисляет проценты и выводит результат. Полностью вывод программы выглядит следующим образом:

```
Счет 1001 - $200.00
Счет 1002 - $112.50 (12.500%)
Нажмите <Enter> для завершения программы...
```


СОВЕТ

Обратите внимание на то, что метод `InitSavingsAccount()` вызывает метод `InitBankAccount()`, который инициализирует члены-данные банковского счета. Метод `InitSavingsAccount()` мог бы делать это непосредственно, однако лучше позволить классу `BankAccount` самостоятельно инициализировать свои члены. Каждый класс должен отвечать за себя сам.

ЯВЛЯЕТСЯ или СОДЕРЖИТ

Отношения между `SavingsAccount` и `BankAccount` представляют собой фундаментальное отношение ЯВЛЯЕТСЯ, которое присуще наследованию. Чуть позже будет рассмотрено альтернативное отношение СОДЕРЖИТ.

Отношение ЯВЛЯЕТСЯ

Отношение ЯВЛЯЕТСЯ (IS_A) между `SavingsAccount` и `BankAccount` можно продемонстрировать путем следующего изменения в классе `Program` демонстрационной программы `SimpleSavingsAccount` из предыдущего раздела.

```
public class Program
{
    // Добавим:
    // DirectDeposit - автоматический вклад на счет
    public static void DirectDeposit(BankAccount ba,
                                     decimal pay)
    {
        ba.Deposit(pay);
    }

    public static void Main(string[] args)
    {
        // Создание банковского счета и вывод на экран
        BankAccount ba = new BankAccount();
        ba.InitBankAccount(100M);
        DirectDeposit(ba,100M);
        Console.WriteLine("Счет {0}",
                          ba.ToBankAccountString());
        // Теперь - депозитный счет
        SavingsAccount sa = new SavingsAccount();
        sa.InitSavingsAccount(100, 12.5M);
```

```
        DirectDeposit(sa,100M);
        sa.AccumulateInterest();
        Console.WriteLine("Счет {0}",
                          sa.ToSavingsAccountString());

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
}
```

Почти ничего не изменилось. Единственное реальное отличие заключается в том, что теперь все вклады делаются с помощью локального метода `DirectDeposit()`, который не является частью класса `BankAccount`. Аргументами этого метода являются банковский счет и величина вклада.

Обратите внимание на то, что метод `Main()` может передать методу `DirectDeposit()` как обычный банковский счет, так и депозитный, поскольку `SavingsAccount` ЯВЛЯЕТСЯ `BankAccount` и, таким образом, имеет все права и привилегии последнего. Поскольку `SavingsAccount` ЯВЛЯЕТСЯ `BankAccount`, вы можете присвоить `SavingsAccount` переменной типа `BankAccount` либо использовать его в качестве аргумента `BankAccount`.

Доступ к BankAccount через содержание

Класс `SavingsAccount` может получить доступ к членам `BankAccount` и другим способом, как показано в приведенном далее фрагменте кода (ключевая строка здесь выделена полужирным шрифтом).

```
// SavingsAccount - банковский счет с накоплением процентов
public class SavingsAccount_ // Обратите внимание на
                             // подчеркивание: это не класс
{                            // SavingsAccount.
    public BankAccount _bankAccount; // Содержит BankAccount
    public decimal _interestRate;
    // InitSavingsAccount - использует процентную ставку,
    // выражаемую числом от 0 до 100
    public void InitSavingsAccount(BankAccount bankAccount,
                                   decimal interestRate)
    {
        _bankAccount = bankAccount;
        _interestRate = interestRate / 100;
    }
    // AccumulateInterest - выполняется однократно при
    // начислении процентов
    public void AccumulateInterest()
    {
```

```
        _bankAccount._balance = _bankAccount.Balance
                    + (_bankAccount.Balance * _interestRate);
    }
    // Deposit - разрешен любой положительный вклад
    public void Deposit(decimal amount)
    {
        // Делегирование содержащемуся объекту BankAccount
        _bankAccount.Deposit(amount);
    }
    // Withdraw - можно снять не более того, что имеется на
    // счету; метод возвращает снятую сумму
    public double Withdraw(decimal withdrawal)
    {
        return _bankAccount.Withdraw(withdrawal);
    }
}
```

В этом случае класс `SavingsAccount_` *содержит* член-данные `_bankAccount` (вместо наследования от `BankAccount`). Объект `_bankAccount` включает номер счета и баланс, необходимые для функционирования `SavingsAccount_`. Класс `SavingsAccount_` содержит данные, специфичные для депозитного счета, и *делегирует* при необходимости запросы к содержащемуся в нем объекту `BankAccount` (т.е. когда классу `SavingsAccount_` нужен, например, баланс, он запрашивает его у содержащегося в нем объекта `BankAccount`).

В этом случае речь идет о том, что `SavingsAccount_` СОДЕРЖИТ `BankAccount`. Иногда говорят о том, что `SavingsAccount` включает (composes) `BankAccount`, т.е. `SavingsAccount` частично состоит из `BankAccount`.

Отношение СОДЕРЖИТ

Отношение СОДЕРЖИТ фундаментально отличается от отношения ЯВЛЯЕТСЯ. Это отличие кажется не столь существенным в следующем фрагменте исходного текста:

```
// Создание нового депозитного счета
BankAccount ba = new BankAccount()
// Особая версия SavingsAccount:
SavingsAccount_ sa = new SavingsAccount_();
sa.InitSavingsAccount(ba, 5);
// Вкладываем 100 на счет
sa.Deposit(100M);
// Подсчитываем проценты
sa.AccumulateInterest();
```

Проблема в том, что теперь `SavingsAccount_` не может использоваться в качестве `BankAccount`, поскольку не является его наследником. Он теперь *содержит* `BankAccount`, а это далеко не одно и то же. Например, следующий код компилироваться не будет:

```
// DirectDeposit - автоматический вклад на счет
void DirectDeposit(BankAccount ba, int pay)
{
    ba.Deposit(pay);
}
void SomeFunction()
{
    // Этот код не скомпилируется
    SavingsAccount_ sa = new SavingsAccount_();
    DirectDeposit(sa, 100);
    // . . . продолжение . . .
}
```

Метод `DirectDeposit()` не может принять `SavingsAccount_` вместо `BankAccount`. Между этими классами нет такого очевидного отношения, как в случае наследования. Тем не менее не следует считать, что содержание — плохая идея, просто для разных ситуаций подходят разные решения.

Когда использовать отношение ЯВЛЯЕТСЯ и когда — СОДЕРЖИТ

Различие между отношениями ЯВЛЯЕТСЯ и СОДЕРЖИТ гораздо глубже, чем просто предмет программного соглашения. Эти отношения проистекают из отношений в реальном мире.

Например, "Запорожец" ЯВЛЯЕТСЯ автомобилем. Автомобиль СОДЕРЖИТ мотор. Если ваш знакомый скажет, чтобы вы заехали за ним на автомобиле, и вы приедете на "Запорожце", ему будет не на что пожаловаться — вы приехали на автомобиле. Но если вы притащите к нему двигатель от "Запорожца", у него будут все основания обидеться на глупую шутку. Класс `Zaporozhets` должен расширять класс `Car` не только для того, чтобы получить доступ к методам `Car`, но и чтобы выразить фундаментальные отношения между этими классами.

К сожалению, начинающий программист может унаследовать `Car` от `Motor` как простейший способ получения доступа к членам `Motor`, которые нужны классу `Car` для управления. Например, `Car` может унаследовать у класса `Motor` метод `Go()`. Однако этот пример вскрывает одну из проблем, возникающих при таком подходе. Несмотря на то что "поехали" звучит одинаково и в машине, и даже в ракете, "поехали" по отношению к машине — это совсем не то, что "поехали" по отношению к мотору. Для того чтобы поехала машина, надо обязательно завести мотор, но это далеко не одно и то же, ведь для того, чтобы поехала машина, надо еще отпустить тормоз, переключиться на первую передачу, отпустить сцепление и т.д. Словом, автомобиль просто не является видом мотора, и этого достаточно.

ЗАПОМНИ!

Элегантность программного обеспечения — это не просто эстетический фактор. Она способствует пониманию кода, повышает его надежность, облегчает поддержку и снижает количество возможных ошибок.

СОВЕТ

Специалисты в области объектно-ориентированного программирования для простоты дизайна рекомендуют отдавать предпочтение отношению СОДЕРЖИТ. Однако, когда это имеет смысл, надо без колебаний применять наследование.

Поддержка наследования в C#

Язык C# реализует ряд возможностей, разработанных для поддержки наследования.

Заменяемость классов

Программа может использовать объект подкласса там, где вызов осуществляется для объекта базового класса. Вы уже могли видеть эту концепцию в одном из примеров. `SomeMethod()` может передавать объект `SavingsAccount` в метод `DirectDeposit()`, который ожидает объект `BankAccount`. Вы можете сделать это преобразование более явным:

```
BankAccount ba;
SavingsAccount sa = new SavingsAccount();
                        // Верно:
ba = sa;                    // Неявное преобразование в
                            // базовый класс разрешено
ba = (BankAccount)sa;       // Но явное преобразование
                            // предпочтительнее
                        // Ошибка:
sa = ba;                    // Неявное преобразование в
                            // подкласс запрещено, но
sa = (SavingsAccount)ba; // явное - допустимо
```

В первой строке объект `SavingsAccount` сохраняется в переменной типа `BankAccount`. C# выполняет необходимое преобразование вместо вас. Во второй строке явным образом использован оператор приведения типа.

Последние две строки преобразуют объект типа `BankAccount` в `SavingsAccount`. Вы можете выполнить эту операцию явно, но C# не сделает ее вместо вас. Это все равно что пытаться преобразовать больший числовой тип, такой как `double`, в меньший, такой как `float`. C # не делает этого неявно, потому что это может приводить к потере данных.

ЗАПОМНИ!

Отношение ЯВЛЯЕТСЯ не рефлексивно. Следовательно, несмотря на то, что "Запорожец" является автомобилем, автомобиль — не обязательно "Запорожец". Аналогично `BankAccount` — не обязательно `SavingsAccount`, так что неявное преобразование в этом направлении запрещено. Последняя строка разрешена, поскольку в ней программист явно указывает, что он берет на себя ответственность за выполнение данного преобразования.

Неверное преобразование времени выполнения

В общем случае приведение объекта от типа `BankAccount` к типу `SavingsAccount` — достаточно опасная операция. Рассмотрим следующий пример:

```
public static void ProcessAmount(BankAccount bankAccount)
{
    // Вносим на счет большую сумму
    bankAccount.Deposit(10000.00M);
    // Если объект - SavingsAccount, добавляем проценты
    SavingsAccount savingsAccount = (SavingsAccount)bankAccount;
    savingsAccount.AccumulateInterest();
}
public static void TestCast()
{
    SavingsAccount sa = new SavingsAccount();
    ProcessAmount(sa);
    BankAccount ba = new BankAccount();
    ProcessAmount(ba);
}
```

Метод `ProcessAccount()` выполняет несколько операций, включая вызов метода `AccumulateInterest()`. Приведение `ba` к типу `SavingsAccount` необходимо, поскольку объект `ba` объявлен как `BankAccount`. Программа корректно компилируется, так как все преобразования типов выполнены явно.

Все нормально работает при первом вызове `ProcessAccount()` из `Test()`. Объект `SavingsAccount` передается методу `ProcessAccount()`. Преобразование типа из `BankAccount` в `SavingsAccount` не вызывает проблем, поскольку объект `ba` изначально был объектом типа `SavingsAccount`.

Однако со вторым вызовом `ProcessAccount()` не все так гладко. Преобразование в тип `SavingsAccount` не может быть разрешено. Объект `ba` не имеет метода `AccumulateInterest()`.

ВНИМАНИЕ!

Некорректное преобразование типов генерирует ошибку в процессе выполнения программы (так называемую *ошибку времени выполнения* (run-time error)). Ошибки времени выполнения гораздо сложнее найти и исправить, чем ошибки времени компиляции. Что еще более

неприятно, такая ошибка может произойти не с вами, а с другим пользователем программы. Обычно особого восторга у пользователей такие ошибки не вызывают.

Избегание неверных преобразований с помощью оператора `is`

Метод `ProcessAccount()` работал бы корректно, если бы мог убедиться, что переданный ему объект действительно имеет тип `SavingsAccount`, перед тем как выполнять преобразование. C# предоставляет для этого два ключевых слова — `is` и `as`.

Оператор `is` получает объект в качестве левого аргумента и тип — в качестве правого. Оператор возвращает значение `true`, если тип времени выполнения объекта слева совместим с типом справа. Этот оператор можно использовать для проверки корректности преобразования перед его выполнением. Предыдущий пример можно модифицировать с применением оператора `is`, что позволит избежать ошибки времени выполнения.

```
public static void ProcessAmount(BankAccount bankAccount)
{
    // Вносим на счет большую сумму
    bankAccount.Deposit(10000.00M);
    // Если объект - SavingsAccount...
    if (bankAccount is SavingsAccount)
    {
        // ... добавляем проценты (преобразование типов
        // гарантированно работает)
        SavingsAccount savingsAccount =
                        (SavingsAccount)bankAccount;
        savingsAccount.AccumulateInterest();
    }
    // В противном случае преобразование не выполняется.
    // Однако почему BankAccount - это не то, чего вы ожидали?
    // Возможно, это какая-то ошибочная ситуация?
}

public static void TestCast()
{
    SavingsAccount sa = new SavingsAccount();
    ProcessAmount(sa);
    BankAccount ba = new BankAccount();
    ProcessAmount(ba);
}
```

Добавление инструкции `is` дает гарантию, что преобразование будет выполнено, только если объект `bankAccount` в действительности имеет тип `SavingsAccount`. При первом вызове метода `ProcessAmount()` оператор `is` вернет значение `true`, но при втором вызове, когда в метод будет передан объект

BankAccount, оператор is вернет false, что позволит избежать некорректного преобразования типов. Такая версия программы не генерирует ошибку времени выполнения.


СОВЕТ

С одной стороны, я настоятельно рекомендую вам защищать все выполняемые преобразования оператором is во избежание возможных ошибок времени выполнения. С другой стороны, я рекомендую избегать приведения типов вообще.

Избегание неверных преобразований с помощью оператора as

Оператор as работает несколько иначе, чем оператор is. Вместо возврата значения типа bool он преобразует объект слева от себя в тип справа, но при этом возвращает null, если такое преобразование некорректно, — вместо генерации ошибки времени выполнения при использовании обычного преобразования. Так что вы всегда должны проверять, не равен ли результат работы оператора as ссылке null:

```
SavingsAccount savingsAccount =
               bankAccount as SavingsAccount;
if(savingsAccount != null)
{
    // Продолжаем работу с использованием savingsAccount
}
// В противном случае мы не можем использовать этот объект и
// должны сгенерировать сообщение об ошибке самостоятельно
```


ЗАПОМНИ!

В общем случае следует предпочитать оператор as как более эффективный. Он сразу выполняет преобразование, в то время как оператор is требует двух этапов: проверки с его использованием и последующего преобразования типа.


ВНИМАНИЕ!

К сожалению, as не работает с переменными типов-значений, так что вы не можете применять его с такими типами, как int, long, double и подобными. В этом случае предпочтительнее использовать оператор is.

Класс object

Рассмотрим следующие связанные классы:

```
public class MyBaseClass {}
public class MySubClass : MyBaseClass {}
```

Соотношение между этими двумя классами позволяет программисту сделать следующую проверку времени выполнения:

```
public class Test
{
    public static void GenericMethod(MyBaseClass mc)
    {
        // Если объект действительно является подклассом
        MySubClass msc = mc as MyBaseClass;
        if(msc != null)
        {
            // ... то и обрабатываем его как подкласс
            // ... продолжение ...
        }
    }
}
```

В этом случае метод `GenericMethod()` в состоянии различить подклассы класса `MyBaseClass` с помощью оператора `as`.

ЗАПОМНИ!

Чтобы помочь различить два не связанных между собой класса с использованием оператора `as`, C# производит все классы от одного общего предка — базового класса `object`. Таким образом, любой класс, который явно не наследует другой класс, наследует класс `object`. А значит, два следующих выражения объявляют классы с одним и тем же базовым классом `object`:

```
class MyClass1 : object {}
class MyClass1 {}
```

Общий базовый класс `object` позволяет написать следующий обобщенный метод:

```
public class Test
{
    public static void GenericMethod(object o)
    {
        MyClass1 mc1 = o as MyClass1;
        if(mc1 != null)
        {
            // Используем объект mc1, полученный преобразованием
            // ...
        }
    }
}
```

Метод `GenericMethod()` может быть вызван для объекта любого типа. Ключевое слово `as` может конвертировать `o` в `MyClass1` из `object`.

Наследование и конструктор

Программа `InheritanceExample`, с которой вы встречались ранее в этой главе, применяет методы `Init...()` для инициализации объектов `BankAccount` и `SavingsAccount` и приведения их в корректное состояние. Оснащение этих классов конструкторами — это, определенно, правильное решение, хотя и со своими сложностями. В следующих разделах показано, как преодолеть проблемы, связанные с использованием методов `Init...()`.

Вызов конструктора по умолчанию базового класса

Когда создается подкласс, всякий раз вызывается конструктор по умолчанию базового класса. Конструктор подкласса автоматически вызывает конструктор базового класса, что видно на примере приведенной далее демонстрационной программы.

```
using System;
// InheritingAConstructor - демонстрация автоматического
// вызова конструктора по умолчанию базового класса
namespace InheritingAConstructor {
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Создание объекта BaseClass");
            BaseClass bc = new BaseClass();

            Console.WriteLine("\nСоздание объекта SubClass");
            SubClass sc = new SubClass();

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
    public class BaseClass
    {
        public BaseClass()
        {
            Console.WriteLine("Конструктор BaseClass");
        }
    }
    public class SubClass : BaseClass
    {
        public SubClass()
        {
            Console.WriteLine("Конструктор SubClass");
        }
    }
}
```

Конструкторы BaseClass и SubClass не делают ничего, кроме вывода строки на экран. Создание объекта BaseClass приводит к вызову конструктора по умолчанию BaseClass. Создание объекта SubClass приводит к вызову конструктора по умолчанию BaseClass перед тем, как вызывается собственный конструктор SubClass. Это ясно видно из вывода рассмотренной демонстрационной программы на экран.

```
Создание объекта BaseClass
Конструктор BaseClass

Создание объекта SubClass
Конструктор BaseClass
Конструктор SubClass
Нажмите <Enter> для завершения программы...
```

ЗАПОМНИ!

Иерархия наследуемых классов весьма напоминает этажи здания. Каждый класс, построенный на основе другого класса, представляет собой новый, верхний этаж. То же самое относится и к конструкторам классов: прежде чем будет вызван конструктор верхнего этажа для его построения, надо построить нижний этаж. Очевидна и причина этого: каждый класс сам отвечает за себя, а значит, подкласс не должен отвечать за инициализацию членов базового класса. BaseClass должен получить возможность сконструировать свои члены до того, как члены SubClass смогут к ним обратиться. Лошадь нужно ставить перед телегой.

Передача аргументов конструктору базового класса

Подкласс вызывает конструктор по умолчанию базового класса, если только не указано иное, — даже из конструктора подкласса, не являющегося конструктором по умолчанию. Вот немного исправленная демонстрационная программа, иллюстрирующая сказанное:

```
using System;
namespace Example
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Вызов SubClass()");
            SubClass sc1 = new SubClass();

            Console.WriteLine("\nВызов SubClass(int)");
            SubClass sc2 = new SubClass(0);

            // Ожидаем подтверждения пользователя
```

```
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
    public class BaseClass
    {
        public BaseClass()
        {
            Console.WriteLine("Конструктор BaseClass " +
                              "(по умолчанию)");
        }
        public BaseClass(int i)
        {
            Console.WriteLine("Конструктор BaseClass (int)");
        }
    }
    public class SubClass : BaseClass
    {
        public SubClass()
        {
            Console.WriteLine("Конструктор SubClass " +
                              "(по умолчанию)");
        }
        public SubClass(int i)
        {
            Console.WriteLine("Конструктор SubClass (int)");
        }
    }
}
```

Выполнение программы приводит к следующему выводу на экран:

```
Вызов SubClass()
Конструктор BaseClass (по умолчанию)
Конструктор SubClass (по умолчанию)

Вызов SubClass(int)
Конструктор BaseClass (по умолчанию)
Конструктор SubClass (int)
Нажмите <Enter> для завершения программы...
```

Данная демонстрационная программа сперва создает объект по умолчанию. Как и ожидалось, C# выполняет конструктор по умолчанию `SubClass`, который сначала передает управление конструктору по умолчанию `BaseClass`. Затем программа создает объект, передавая целочисленный аргумент. Как и предполагалось, теперь C# вызывает конструктор `SubClass(int)`. Этот конструктор, в свою очередь, вызывает конструктор по умолчанию `BaseClass`, как и в предыдущем примере, поскольку никакие данные базовому классу не передаются.

Указание конкретного конструктора базового класса

Конструктор подкласса может вызвать определенный конструктор базового класса с использованием ключевого слова `base`. Эта возможность аналогична способу, которым один конструктор может вызвать другой конструктор того же класса с применением ключевого слова `this`. Рассмотрим, например, следующую демонстрационную программу `InvokeBaseConstructor`:

```
// InvokeBaseConstructor - демонстрация того, как подкласс
// может вызвать конструктор базового класса по своему
// выбору с использованием ключевого слова base
using System;

namespace InvokeBaseConstructor
{
    public class BaseClass
    {
        public BaseClass()
        {
            Console.WriteLine("Конструктор BaseClass " +
                              "(по умолчанию)");
        }
        public BaseClass(int i)
        {
            Console.WriteLine("Конструктор BaseClass({0})", i);
        }
    }

    public class SubClass : BaseClass
    {
        public SubClass()
        {
            Console.WriteLine("Конструктор SubClass " +
                              "(по умолчанию)");
        }
        public SubClass(int i1, int i2) : base(i1)
        {
            Console.WriteLine("Конструктор SubClass({0}, {1})",
                              i1, i2);
        }
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Вызов SubClass()");
            SubClass sc1 = new SubClass();
            Console.WriteLine("\nВызов SubClass(1, 2)");
            SubClass sc2 = new SubClass(1, 2);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
```

```
            Console.Read();
        }
    }
}
```

Вывод программы выглядит следующим образом:

```
Вызов SubClass()
Конструктор BaseClass (по умолчанию)
Конструктор SubClass (по умолчанию)

Вызов SubClass(1, 2)
Конструктор BaseClass(1)
Конструктор SubClass(1, 2)
Нажмите <Enter> для завершения программы...
```

Эта версия демонстрационной программы начинается так же, как и предыдущие примеры, — с создания объекта `SubClass` с применением конструкторов по умолчанию как для класса `SubClass`, так и для класса `BaseClass`.

Второй объект создается с помощью выражения `new SubClass(1,2)`. C# вызывает конструктор `SubClass(int,int)`, в котором используется ключевое слово `base` для передачи одного из значений конструктору `BaseClass(int)`. `SubClass` передает первый аргумент для обработки базовому классу, а со вторым работает самостоятельно.

Обновленный класс BankAccount

Демонстрационная программа `ConstructorSavingsAccount` представляет собой обновленную версию демонстрационной программы `SimpleBankAccount`. В этой версии конструктор `SavingsAccount` может передавать информацию конструктору `BankAccount`. Здесь приведены только метод `Main()` и указанные конструкторы.

```
// ConstructorSavingsAccount - реализует SavingsAccount как
// вид BankAccount; не использует виртуальные методы, но
// корректно реализует конструкторы
using System;
namespace ConstructorSavingsAccount {
    // BankAccount - модель банковского счета с номером счета
    // (назначаемым при создании) и балансом
    public class BankAccount
    {
        // Номера счетов начинаются с 1000 и последовательно
        // увеличиваются
        public static int _nextAccountNumber = 1000;

        // Номер счета и баланс для каждого объекта
```

```
    public int _accountNumber;
    public decimal _balance;

    // Конструкторы
    public BankAccount() : this(0) {
    }
    public BankAccount(decimal initialBalance) {
        _accountNumber = ++_nextAccountNumber;
        _balance = initialBalance;
    }
    public decimal Balance
    {
        get { return _balance; }
        // Защищенная функция доступа позволяет подклассам
        // использовать свойство Balance для установки значений
        protected set { _balance = value; }
    }
    // Deposit - разрешен любой положительный вклад
    public void Deposit(decimal amount) {
        if (amount > 0) {
            Balance += amount;
        }
    }
    // Withdraw - можно снять не более того, что имеется
    // на счету; метод возвращает снятую сумму
    public decimal Withdraw(decimal withdrawal) {
        if (Balance <= withdrawal) {
            withdrawal = Balance;
        }

        Balance -= withdrawal;
        return withdrawal;
    }
    // ToString - строка с информацией о состоянии счета
    public string ToBankAccountString() {
        return String.Format("{0} - {1:C}",
                             _accountNumber, Balance);
    }
}

// SavingsAccount - банковский счет с начислением
// процентов
public class SavingsAccount : BankAccount
{
    public decimal _interestRate;
    // InitSavingsAccount - использует процентную ставку,
    // выражаемую числом от 0 до 100
    public SavingsAccount(decimal interestRate)
    : this(interestRate, 0) { }

    public SavingsAccount(decimal interestRate, decimal initial)
    : base(initial) { this._interestRate = interestRate / 100; }
```

```
        // AccumulateInterest - вызывается однократно в конце
        // периода начисления процентов
        public void AccumulateInterest() {
            // Использование защищенных методов доступа
            // с помощью свойства Balance
            Balance = Balance + (decimal)(Balance * _interestRate);
        }
        // ToString - строка с информацией о состоянии счета
        public string ToSavingsAccountString() {
            return String.Format("{0} ({1}%)",
                    ToBankAccountString(), _interestRate * 100);
        }
    }

    public class Program
    {
        // DirectDeposit - автоматический внос денег на счет
        public static void DirectDeposit(BankAccount ba,
                                         decimal pay)
        {
            ba.Deposit(pay);
        }
        public static void Main(string[] args)
        {
            // Создание банковского счета и вывода
            // информации о нем
            BankAccount ba = new BankAccount(100M);
            DirectDeposit(ba, 100M);
            Console.WriteLine("Счет {0}",
                              ba.ToBankAccountString());

            // То же для счета с накоплением процентов
            SavingsAccount sa = new SavingsAccount(12.5M);
            DirectDeposit(sa, 100M);
            sa.AccumulateInterest();
            Console.WriteLine("Счет {0}",
                              sa.ToSavingsAccountString());

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

Класс `BankAccount` определяет два конструктора: один, который получает начальное значение баланса, и второй — конструктор по умолчанию, не получающий никаких аргументов. Чтобы избежать дублирования кода конструкторов, конструктор по умолчанию вызывает конструктор с передаваемым начальным значением баланса посредством ключевого слова `this`.

Класс `SavingsAccount` также предоставляет в распоряжение программиста два конструктора. Конструктор `SavingsAccount`, принимающий в качестве аргумента величину процентной ставки, вызывает конструктор `SavingsAccount`, принимающий в качестве аргументов величину процентной ставки и начальное значение баланса, передавая в качестве последнего 0. В свою очередь, этот конструктор наиболее общего вида передает начальное значение баланса соответствующему конструктору `BaseClass` (все это отражено на диаграмме на рис. 16.1).

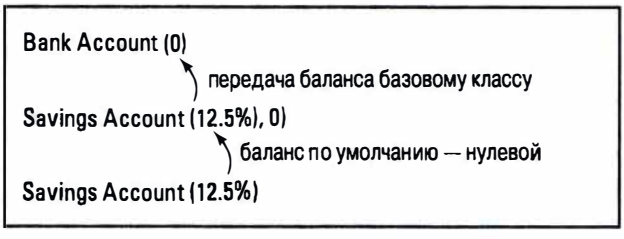


Рис. 16.1. Передача параметров в цепочке вызовов конструкторов

Программа модифицирована таким образом, чтобы избежать вызова внутренних методов `Init...()`, заменив их автоматически вызываемыми конструкторами. Вывод этой демонстрационной программы ничем не отличается от вывода ее предшественницы.

СОВЕТ

Обратите внимание на свойство `Balance` класса `BankAccount`, которое представляет собой открытый метод доступа для получения значения, но защищенный — для его установки. Применение ключевого слова `protected` предотвращает использование этого метода извне класса `BankAccount`, но разрешает при этом его применение наследникам данного класса, например в методе `SavingsAccount.AccumulateInterest`, где свойство `Balance` находится слева от оператора присваивания. (Свойства и ключевое слово `protected` рассматриваются в главе 15, "Класс: каждый сам за себя".)

СБОРКА МУСОРА И ДЕСТРУКТОРЫ C#

Язык C# предоставляет метод, обратный конструктору и именуемый *деструктором*. Деструктор имеет то же имя, что и имя класса, но предваренное символом тильды (~). Например, метод `~BaseClass()` является деструктором класса `BaseClass()`.

Язык C# вызывает деструктор, когда перестает использовать объект. Класс может иметь только деструктор по умолчанию (не имеющий параметров),

поскольку деструктор не вызывается явно. Кроме того, деструктор всегда виртуален.

При использовании наследования деструкторы вызываются в порядке, обратном порядку вызова конструкторов. Таким образом, деструктор подкласса вызывается перед деструктором базового класса.

Деструктор в C# гораздо менее полезен, чем в ряде других объектно-ориентированных языков программирования, таких как C++, поскольку в C# используется *недетерминированная деструкция*. Этот термин и его важность требуют определенных пояснений.

Память для объекта выделяется из кучи при выполнении команды `new`, например `new SubClass()`. Блок памяти остается зарезервированным до тех пор, пока имеется хоть одна корректная ссылка на эту память. Вы можете иметь несколько переменных, ссылающихся на один и тот же объект.

О памяти говорят, что она *недостижима*, когда из области видимости выходит последняя ссылка на нее. Другими словами, никто не в состоянии обратиться к блоку памяти после утраты последней ссылки на нее. Когда блок памяти становится недостижимым, C# не предпринимает никаких конкретных действий. В фоновом режиме выполняется низкоприоритетный системный процесс, который проводит поиск недостижимых блоков памяти. Такой "сборщик мусора" запускается, когда в работе программы наступает затишье, чтобы не повлиять отрицательно на ее производительность. Когда сборщик мусора находит недостижимый блок памяти, он возвращает его в кучу.

Обычно сборщик мусора незаметно работает в фоновом режиме и получает управление только на короткие периоды времени, когда начинает чувствоваться нехватка памяти.

Деструкторы C#, такие как `~BaseClass()`, являются недетерминированными, поскольку не вызываются до тех пор, пока объект не будет подобран сборщиком мусора, а это может случиться через продолжительное время после того, как объект перестанет использоваться. Может даже возникнуть ситуация, когда программа завершится до того, как будет выполнена очередная сборка мусора, и в этом случае деструктор не будет вызван вообще. *Недетерминированный* означает, что вы не можете предсказать, когда объект будет уничтожен сборщиком мусора. Может пройти немало времени до того, как объект будет подобран сборщиком мусора и будет вызван деструктор этого объекта.

Основной вывод — программисту на C# использовать деструкторы приходится очень редко. В C# имеются другие способы вернуть системе захваченные ресурсы, которые больше не нужны, — с применением метода `Dispose()`, изучение которого, увы, выходит за рамки настоящей книги (вы можете прочесть о нем в разделе *Dispose method* справочной системы C#).

Глава 17

Полиморфизм

В ЭТОЙ ГЛАВЕ...

- Скрывать или перекрывать методы базового класса?
- Создание абстрактных классов и методов
- Применение `ToString`
- Защита класса от наследования

Наследование позволяет одному классу "приспособить" члены другого класса. Таким образом, можно создать класс `SavingsAccount`, который наследует члены-данные и методы, такие как `Deposit()`, от базового класса `BankAccount`. Это полезно, но этого недостаточно для имитации объектов реального мира. (Если вы не знаете или забыли, что такое наследование классов, читайте главу 16, "Наследование".)

Микроволновая печь представляет собой определенный тип печи, но не из-за внешнего вида, а потому что она выполняет те же функции, что и любая печь. Она может выполнять и ряд дополнительных функций, но как минимум она должна реализовать базовую функцию печи — подогревать еду. При этом вас не должно беспокоить, что у нее внутри, кто ее сделал и как продавец сумел-таки всучить ее вашей жене по такой цене на распродаже...

С точки зрения обычного потребителя, отличия микроволновой печи от обычной не так важны — лишь бы они обе могли готовить любимые блюда, но если взглянуть на это с точки зрения печи, то эти отличия становятся крайне существенны, поскольку внутреннее устройство печей совершенно различно.

ЗАПОМНИ!

Мощь наследования заключается в том факте, что подкласс *не обязан* наследовать каждый метод базового класса в том виде, в котором он написан. Подкласс может наследовать суть метода базового класса при полном отличии его реализации.

Перегрузка унаследованного метода

Как описано в главе 3, "Работа со строками", несколько методов могут иметь одинаковые имена, лишь бы различались количества и/или типы их аргументов.

Простейший случай перегрузки метода

ЗАПОМНИ!

Два метода с одинаковыми именами называются *перегруженными* (overloaded). Аргументы метода становятся частью его расширенного имени (используемого C# внутренне), как показано в следующем фрагменте исходного текста:

```
public class MyClass
{
    public static void AMethod()
    {
        // Некоторые действия
    }
    public static void AMethod(int)
    {
        // Некоторые другие действия
    }
    public static void AMethod(double d)
    {
        // Некоторые действия, отличные от первых двух
    }
    public static void Main(string[] args)
    {
        AMethod();
        AMethod(1);
        AMethod(2.0);
    }
}
```

Язык C# в состоянии различать эти методы по их аргументам. Каждый из вызовов в методе `Main()` обращается к своему методу.

ЗАПОМНИ!

Возвращаемый тип не является частью расширенного имени метода, так что вы не можете иметь два метода, различающиеся только типами возвращаемого значения.

Различные классы, различные методы

Не удивительно, что класс, которому принадлежит метод, также становится частью его расширенного имени. Рассмотрим следующий фрагмент исходного текста:

```
public class MyClass
{
    public static void AMethod1();
    public void AMethod2();
}

public class UrClass
{
    public static void AMethod1();
    public void AMethod2();
}

public class Program
{
    public static void Main(string[] args)
    {
        UrClass.AMethod1(); // Вызов статического метода

        // Вызов метода-члена MyClass.AMethod2()
        MyClass mcObject = new MyClass();
        mcObject.AMethod2();
    }
}
```

Имя класса является частью расширенного имени метода, так что для C# метод `MyClass.AMethod1()` не имеет ничего общего с методом `UrClass.AMethod1()`.

Сокрытие метода базового класса

Итак, метод одного класса может перегружать другой метод того же класса, если использует другие аргументы. Как оказывается, метод может также перегружать метод базового класса. Перегрузка метода базового класса известна как *сокрытие* (hiding) метода.

Предположим, ваш банк проводит новую политику, в соответствии с которой снятие с депозитного счета отличается от других типов снятия со счета. Предположим для конкретности, что каждое снятие со счета обходится вкладчику в 1,50 доллара.

При использовании функционального подхода вы можете реализовать эту политику посредством переменной-флага в классе, который указывал бы, принадлежит объект типу `SavingsAccount` или `BankAccount`. В этом случае метод снятия со счета должен проверять значение флага, чтобы выяснить, следует ли

снимать дополнительные 1.50, как показано в следующем фрагменте исходного текста.

```
public class BankAccount
{
    private decimal _balance;
    private bool _isSavingsAccount;
    // Начальный баланс и флаг, указывающий,
    // является ли счет депозитным
    public BankAccount(decimal initialBalance,
                       bool isSavingsAccount)
    {
        _balance = initialBalance;
        this._isSavingsAccount = isSavingsAccount;
    }
    public decimal Withdraw(decimal amountToWithdraw)
    {
        // Если счет депозитный . . .
        if (_isSavingsAccount)
        {
            // ...снимаем лишние 1.50
            _balance -= 1.50M;
        }
        // Далее обычный код снятия со счета
        if (amountToWithdraw > _balance)
        {
            amountToWithdraw = _balance;
        }
        _balance -= amountToWithdraw;
        return amountToWithdraw;
    }
}

class MyClass
{
  public void SomeFunction()
  {
    // Создаем депозитный счет
    BankAccount ba = new BankAccount(0, true);
  }
}
```

Ваш метод должен указывать, какой именно счет создается, путем передачи дополнительного аргумента конструктору `BankAccount`. Конструктор сохраняет флаг, затем используемый в методе `Withdraw()` для снятия дополнительной суммы `1.50`.

Объектно-ориентированный подход состоит в сокрытии метода `Withdraw()` базового класса `BankAccount` новым методом с тем же именем в классе `SavingsAccount`, как показано в приведенной далее демонстрационной программе.

```
// HidingWithdrawal - сокрытие метода базового класса
// методом подкласса с тем же именем
using System;

namespace HidingWithdrawal
{
    // BankAccount - базовый банковский счет
    public class BankAccount
    {
        protected decimal _balance;

        public BankAccount(decimal initialBalance)
        {
            _balance = initialBalance;
        }

        public decimal Balance
        {
            get { return _balance; }
        }

        public decimal Withdraw(decimal amount)
        {
            // Хорошая практика состоит в том, чтобы изменять
            // не входные параметры, а их копии
            decimal amountToWithdraw = amount;

            if (amountToWithdraw > Balance)
            {
                amountToWithdraw = Balance;
            }

            _balance -= amountToWithdraw;
            return amountToWithdraw;
        }
    }

    // SavingsAccount - банковский счет с начислением
    // процентов
    public class SavingsAccount : BankAccount
    {
        public decimal _interestRate;

        // SavingsAccount - процентная ставка передается как
        // число от 0 до 100
        public SavingsAccount(decimal initialBalance,
                              decimal interestRate)
            : base(initialBalance)
        {
            _interestRate = interestRate / 100;
        }
```

```
        // AccumulateInterest - начисление процентов один
        // раз за определенный период
        public void AccumulateInterest()
        {
            _balance = Balance + (Balance * _interestRate);
        }

        // Withdraw - со счета можно снять любую сумму, не
        // превышающую баланс; метод возвращает снятую сумму
        public decimal Withdraw(decimal withdrawal)
        {
            // Дополнительное снятие 1.50
            base.Withdraw(1.5M);
            // Теперь снимаем со счета как обычно
            return base.Withdraw(withdrawal);
        }
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            BankAccount ba;
            SavingsAccount sa;
            // Создаем банковский счет, снимаем 100, выводим
            // результат
            ba = new BankAccount(200M);
            ba.Withdraw(100M);
            // Делаем то же с депозитным счетом
            sa = new SavingsAccount(200M, 12);
            sa.Withdraw(100M);
            // Выводим состояния счетов
            Console.WriteLine("Баланс BankAccount равен {0:C}",
                              ba.Balance);
            Console.WriteLine("Баланс SavingsAccount равен {0:C}",
                              sa.Balance);

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

В этом случае метод `Main()` создает объект `BankAccount` с начальным балансом 200 долларов и снимает с него 100 долларов. Затем те же действия выполняются с объектом `SavingsAccount`. Когда метод `Main()` снимает деньги со счета базового класса, метод `BankAccount.Withdraw()` снимает только указанную сумму (но не более суммы на счету). Когда же метод `Main()` снимает деньги с депозитного счета, метод `SavingsAccount.Withdraw()` снимает дополнительную сумму, равную 1,50 доллара.

СОВЕТ

Обратите внимание на то, что метод `SavingsAccount.Withdraw()` использует метод базового класса `BankAccount.Withdraw()`, а не работает непосредственно с балансом. Если это возможно, базовый класс должен сам работать со своими членами-данными.

Чем сокрытие лучше дополнительной проверки

На первый взгляд, добавление флага в метод `BankAccount.Withdraw()` представляется более простым решением, чем предложенный вариант с сокрытием метода базового класса. В конце концов, использование флага потребовало добавления всего лишь четырех строк, две из которых — просто фигурные скобки.

Однако простое решение порождает сложные проблемы. Первая заключается в том, что класс `BankAccount` не должен беспокоиться о деталях работы `SavingsAccount`. Говоря формально, это нарушает принцип инкапсуляции. Базовый класс не должен ничего знать о своих потомках, потому что это ведет к реальной проблеме. Предположим, банк решает добавить новые счета, например `CheckingAccount`, `CDAccount`, `TBillAccount`. У каждого из них — свои правила снятия денег со счета и каждый использует собственный флаг. После трех-четырех добавлений новых типов счетов старый метод `BankAccount.Withdraw()` начинает выглядеть слишком сложным. Каждый новый вид счета приводит ко все б*о*льшим изменениям этого метода. Такое решение совершенно не подходит. Классы должны отвечать сами за себя.

Случайное сокрытие метода базового класса

Как ни странно, метод базового класса может оказаться скрытым просто случайно. Пусть, например, имеется метод `Vehicle.TakeOff()`, который начинает движение транспортного средства. Позже кто-то может расширить класс `Vehicle`, создав класс `Airplane`. Понятно, что метод `TakeOff()` этого класса совершенно иной, чем класса `Vehicle`. Очевидно, что это случай ложной тождественности — два метода не имеют ничего общего, кроме имени. К счастью, C# в состоянии обнаружить эту проблему.

Язык C# генерирует зловещего вида предупреждение при компиляции рассматривавшейся ранее демонстрационной программы `HidingWithdrawal`. Из всего длинного текста предупреждения интерес представляет только небольшая его часть, а именно:

```
'...SavingsAccount.Withdraw(decimal)' hides inherited member
    '...BankAccount.Withdraw(decimal)'.
    Use the new keyword if hiding was intended.
```

Язык C# пытается сообщить, что вы написали в подклассе метод с тем же именем, что и у метода базового класса. Действительно ли вы хотите именно этого?

СОВЕТ

Это всего лишь предупреждение. Вы можете и не реагировать на него, но все же крайне желательно ознакомиться со всеми предупреждениями, выводимыми компилятором, и избавиться от них. Предупреждение почти всегда говорит о какой-то мелочи, которая может перерасти в крупные неприятности, если вовремя о ней не позаботиться.

РАССМАТРИВАЙТЕ ПРЕДУПРЕЖДЕНИЯ КАК ОШИБКИ

Неплохо дать указание компилятору C# рассматривать все предупреждения как ошибки, по крайней мере на этапе отладки. Для этого следует воспользоваться командой меню Project⇨Properties (Проект⇨Свойства) и прокрутить панель построения страницы свойств проекта до раздела Errors and Warnings (Ошибки и предупреждения). Установите значение параметра Warning Level (Уровень предупреждений) равным 4, наивысшей возможной величине. Кроме того, в подразделе Treat Warnings as Errors (Обрабатывать предупреждения как ошибки) выберите флаг All (Все). (Если какое-то предупреждение станет "мозолить глаза" при том, что вы понимаете, о чем речь, его можно будет убрать с глаз долой, поместив в список подавляемых предупреждений.)

При таких установках, работая над программой, вы будете вынуждены устранять все предупреждения так же, как устраняете реальные ошибки. Даже если вы не будете заставлять компилятор считать предупреждения ошибками, оставьте уровень предупреждений равным 4 и тщательно просматривайте весь список предупреждений после каждой сборки программы.

Дескриптор `new`, упомянутый в предупреждении и показанный в приведенном далее фрагменте исходного текста, говорит компилятору C# о том, что сокрытие метода преднамеренное (и тем самым устраняет предупреждение).

```
// Теперь с Withdraw() никаких проблем
new public decimal Withdraw(decimal withdrawal)
{
    // ... Никаких иных изменений не требуется ...
}
```

СОВЕТ

Такое использование ключевого слова `new` не имеет ничего общего с его применением для создания объекта (C# перегружает сам себя!).

Вызов методов базового класса

Вернемся к методу `SavingsAccount.Withdraw()` из демонстрационной программы `HidingWithdrawal`, рассмотренной ранее в этой главе. Вызов

BankAccount.Withdraw() из этого нового метода осуществляется при помощи ключевого слова base. Приведенная далее версия метода без ключевого слова base работать не будет:

```
new public decimal Withdraw(decimal withdrawal)
{
    decimal amountWithdrawn = Withdraw(withdrawal);
    amountWithdrawn += Withdraw(1.5);
    return amountWithdrawn;
}
```

В этом случае возникает та же проблема, что и в следующем фрагменте:

```
void fn()
{
    fn(); // Вызов самого себя
}
```

Вызов fn() из fn() приводит к *рекурсивному* вызову методом самого себя. Аналогично такой вызов Withdraw(), как показано в фрагменте выше, приводит к вызову методом самого себя, пока программа в конечном счете не завершится аварийно.

Требуется некоторым способом указать C#, что в методе SavingsAccount.Withdraw() следует вызвать метод BankAccount.Withdraw(). Один из вариантов решения поставленной задачи состоит в преобразовании указателя this в указатель на объект BankAccount перед выполнением вызова:

```
// Withdraw - эта версия обращается к сокрытому методу
// базового класса посредством явного преобразования this
new public decimal Withdraw(decimal withdrawal)
{
    // Преобразование указателя this в объект класса
    // BankAccount
    BankAccount ba = (BankAccount)this;

    // Вызов Withdraw() с использованием объекта BankAccount
    decimal amountWithdrawn = ba.Withdraw(withdrawal);
    amountWithdrawn += ba.Withdraw(1.5);
    return amountWithdrawn;
}
```

Данное решение вполне работоспособно: ba.Withdraw() вызывает метод класса BankAccount. Однако в будущем изменение программы может привести к такому изменению иерархии классов, что SavingsAccount не будет непосредственным потомком BankAccount. Подобная модификация приведет к неверной работе метода, найти причину которой будет нелегко.

Необходим способ пояснить C#, что требуется вызвать метод Withdraw() из класса, являющегося непосредственным предшественником текущего, причем без явного именования этого класса. Для этой цели в C# служит ключевое слово base.

ЗАПОМНИ!

Это то же ключевое слово `base`, которое конструктор использует для передачи аргумента конструктору базового класса. Ключевое слово C# `base` в показанном далее фрагменте кода подобно ключевому слову `this`, но приведение к базовому классу выполняется независимо от того, какой именно класс является таковым.

```
// Withdraw - можно снимать любую сумму в пределах баланса;
// возвращает снятую со счета сумму
new public decimal Withdraw(decimal withdrawal)
{
    // Снятие дополнительной суммы 1.50
    base.Withdraw(1.5M);
    // Снятие со счета с оставшейся суммой
    return base.Withdraw(withdrawal);
}
```

Вызов `base.Withdraw()` приводит к вызову метода `BankAccount.Withdraw()`; тем самым проблема, связанная с рекурсией, снимается. Кроме того, данное решение работает и при изменении иерархии наследования.

Полиморфизм

Можно перегрузить метод базового класса методом в подклассе. Это и замечательно, и одновременно очень опасно.

Проведем мысленный эксперимент: когда должно приниматься решение о том, какой из методов (`BankAccount.Withdraw()` или `SavingsAccount.Withdraw()`) будет вызван, — во время компиляции или во время выполнения программы? Для того чтобы понять, в чем здесь различие, будет немного изменена рассмотренная ранее программа `HidingWithdrawal` (здесь приведена только та часть, в которую внесены изменения). Вот ее новая версия:

```
// HidingWithdrawalPolymorphically - сокрытие метода
// Withdraw() базового класса методом с тем же именем в
// подклассе
public class Program
{
    public static void MakeAWithdrawal(BankAccount ba,
                                        decimal amount)
    {
        ba.Withdraw(amount);
    }
    public static void Main(string[] args)
    {
        BankAccount ba;
        SavingsAccount sa;
        ba = new BankAccount(200M);
        MakeAWithdrawal(ba, 100M);
```

```
        sa = new SavingsAccount(200M, 12);
        MakeAWithdrawal(sa, 100M);

        // Выводим состояния счетов
        Console.WriteLine("Баланс BankAccount равен {0:C}",
                          ba.Balance);
        Console.WriteLine("Баланс SavingsAccount равен {0:C}",
                          sa.Balance);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
}
```

Вывод этой демонстрационной программы на экран может вас удивить (а может и не удивить — в зависимости от того, чего именно вы ожидали):

```
Баланс BankAccount равен $100.00
Баланс SavingsAccount равен $100.00
Нажмите <Enter> для завершения программы...
```

В этот раз вместо снятия со счета в методе `Main()` программа передает объект счета методу `MakeAWithdrawal()`.

Первый вопрос очевиден: почему метод `MakeAWithdrawal()` может принимать объект `SavingsAccount`, если он ожидает в качестве аргумента объект `BankAccount`? Ответ не менее ясен: потому что `SavingsAccount` ЯВЛЯЕТСЯ `BankAccount` (см. главу 16, "Наследование").

Второй вопрос не так очевиден. Когда методу `MakeAWithdrawal()` передается объект `BankAccount`, он вызывает `BankAccount.Withdraw()`, и это понятно. Но когда передается объект типа `SavingsAccount`, вызывается тот же метод. Должен ли в этом случае вызываться метод `Withdraw()` подкласса?

С одной стороны, поскольку объект `ba` принадлежит типу `BankAccount`, вызов `ba.Withdraw()` должен вызывать метод `BankAccount.Withdraw()`. С другой стороны, хотя объект `ba` и объявлен как `BankAccount`, фактически он представляет собой объект `SavingsAccount`, так что должен быть вызван метод `SavingsAccount.Withdraw()`. Оба аргумента достаточно логичны.

В данном случае C# принимает как более весомый первый аргумент. Это более безопасный выбор — работать с объявленным типом, — поскольку он устраняет все недоразумения. Объект объявлен как `BankAccount`, и так тому и быть.

Что неверно в стратегии использования объявленного типа

В ряде случаев вам не требуется работа с объявленным типом. На самом деле необходимо, чтобы вызов базировался на *реальном типе*, т.е. на типе времени исполнения, а не на объявленном типе. Например, вам нужно, чтобы

выполнялись действия со счетом типа `SavingsAccount`, который хранится в переменной типа `BankAccount`. Такая возможность *принятия решения во время выполнения* программы называется *полиморфизмом* или *поздним связыванием* (late binding). Стратегия использования объявленного типа называется *ранним связыванием* (early binding) в противоположность позднему.

Термин *полиморфизм* происходит из греческого языка: *поли* означает "много", а *морф* — "форма" (или действие).

Следовательно, полиморфизм — это идея или концепция преобразования одного объекта `BankAccount` во множество различных объектов, `BankAccount` или `SavingsAccount` (в данном случае). Полиморфизм и позднее связывание — не совсем одно и то же понятие, но их различие весьма тонкое.

- *Полиморфизм* означает возможность принятия решения о том, какой метод должен быть вызван в процессе выполнения программы.
- *Позднее связывание* — способ реализации полиморфизма языком программирования.

Полиморфизм является ключевой составляющей объектно-ориентированного программирования. Он настолько важен, что языки, его не поддерживающие, не имеют права называться объектно-ориентированными.

Языки программирования, поддерживающие классы, но не поддерживающие полиморфизм, называются *объектно-основанными языками* (object-based languages). Примером такого языка может служить язык Visual Basic 6.0 (но не VB .NET).

Без полиморфизма в наследовании мало толку. Позвольте привести наглядный пример, иллюстрирующий данный тезис. Предположим, вы написали мощную программу, использующую класс... ну, скажем, `Student`. После нескольких месяцев проектирования, кодирования и тестирования вы наконец-то вынесли ее на суд восхищенных пользователей (начинающих даже поговаривать, что совершенно напрасно не существует Нобелевской премии в области программирования).

Проходит время, и ваш шеф требует, чтобы вы добавили в программу возможность работы с аспирантами, которые, конечно, похожи на студентов, но все же немного отличаются от них (сами аспиранты считают, что они отличаются во всем). Предположим, что формулы для вычисления оплаты за обучение для студентов и аспирантов различны. Вашему боссу это безразлично, но в программе имеется масса вызовов метода `CalcTuition()`, предназначенного для таких расчетов. Вот пример одного из таких вызовов:

```
void someMethod(Student s)
{
    // ... Какие-то действия ...
    s.CalcTuition();
    // ... продолжение ...
}
```

Если бы C# не поддерживал позднее связывание, то вам бы пришлось редактировать метод `someMethod()`, чтобы проверять в нем, является ли переданный объект `s` переменной типа `Student` или `GraduateStudent`. Программа должна была бы вызывать `Student.CalcTuition()` в случае, когда переменная `s` принадлежала бы классу `Student`, и `GraduateStudent.CalcTuition()` для класса `GraduateStudent`.

Это было бы не так страшно, если бы не две проблемы.

- Это только один метод. А теперь представьте, что `CalcTuition()` вызывается в сотнях мест...
- Предположим, что `CalcTuition()` — не единственное различие между двумя классами. Шансы, что вы найдете все места в программе, требующие изменений, резко снижаются...

При использовании полиморфизма вы просто позволяете C# самостоятельно решить, какой метод должен быть вызван.

Использование `is` для полиморфного доступа к скрытому методу

Каким образом сделать программу полиморфной? Один из подходов для решения этой задачи в C# состоит в использовании ключевого слова `is` (о котором (и его родственнике `as`) рассказывалось в главе 16, "Наследование"). Выражение `ba is SavingsAccount` возвращает значение `true` или `false` в зависимости от класса объекта во время выполнения программы. Объявленный тип может быть `BankAccount`, но с какого типа объектом приходится иметь дело в реальности? В приведенном далее фрагменте исходного текста `is` используется для обращения к методу `Withdraw()` класса `SavingsAccount`.

```
public class Program
{
    public static void MakeAWithdrawal(BankAccount ba, decimal amount)
    {
        if(ba is SavingsAccount)
        {
            SavingsAccount sa = (SavingsAccount)ba;
            sa.Withdraw(amount);
        }
        else
```

```
        {
            ba.Withdraw(amount);
        }
    }
}
```

Теперь, когда `Main()` передает методу объект типа `SavingsAccount`, метод `MakeAWithdrawal()` проверяет тип времени выполнения объекта `ba` и вызывает метод `SavingsAccount.Withdraw()`.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Программист может выполнить вызов одной строкой:

```
((SavingsAccount)ba).Withdraw(amount); // Обратите внимание на скобки
```

Этот способ часто встречается в программах, написанных опытными разработчиками, которые ненавидят вводить текста больше, чем нужно. Хотя вы можете использовать этот подход, он приводит к менее удобочитаемому исходному тексту, соответственно — к большему количеству ошибок в программе.

Подход с использованием `is` вполне работоспособен, но это плохая идея. Применение `is` требует от метода `MakeAWithdrawal()` осведомленности о всех возможных типах счетов, которые имеются (и могут появиться в дальнейшем) в банке. Это накладывает на метод `MakeAWithdrawal()` слишком большую ответственность. Да, сейчас ваше приложение обходится двумя классами, но завтра от вас могут потребовать реализовать новый вид счета, например `CheckingAccount`, и вы будете вынуждены перерыть всю программу в поисках мест, в которые надо внести добавления, связанные с проверкой типа аргумента метода в процессе выполнения программы.

Объявление метода виртуальным и перекрытие

Как автор метода `MakeAWithdrawal()` вы, конечно, не хотели бы делать его осведомленным обо всех возможных типах счетов. Хотелось бы поручить это программисту, использующему метод `MakeAWithdrawal()`, т.е. заставить C# самостоятельно принимать решение о том, какой метод должен быть вызван, основываясь на информации о типе объекта времени выполнения программы.

Можно заставить C# самостоятельно принимать решение о версии `Withdraw()`, которую следует вызвать. Для этого необходимо пометить метод базового класса при помощи ключевого слова `virtual`, а каждую версию метода в подклассах — ключевым словом `override`.

Следующий пример основан на полиморфизме. В нем в методы `Withdraw()` добавлены инструкции вывода, чтобы доказать, что действительно вызываются правильные методы. Вот код программы `PolymorphicInheritance`:

```
// PolymorphicInheritance - полиморфное сокрытие метода
// базового класса
using System;

namespace PolymorphicInheritance
{
    // BankAccount - простейший банковский счет
    public class BankAccount
    {
        protected decimal _balance;

        public BankAccount(decimal initialBalance)
        {
            _balance = initialBalance;
        }

        public decimal Balance
        {
            get { return _balance; }
        }

        public virtual decimal Withdraw(decimal amount)
        {
            Console.WriteLine("BankAccount.Withdraw() с ${0}...",
                              amount);
            decimal amountToWithdraw = amount;

            if (amountToWithdraw > Balance)
            {
                amountToWithdraw = Balance;
            }

            _balance -= amountToWithdraw;
            return amountToWithdraw;
        }
    }

    // SavingsAccount - банковский счет с начислением
    // процентов
    public class SavingsAccount : BankAccount
    {
        public decimal _interestRate;

        // SavingsAccount - процентная ставка указывается как
        // число от 0 до 100
        public SavingsAccount(decimal initialBalance,
                              decimal interestRate)
            : base(initialBalance)
        {
            _interestRate = interestRate / 100;
        }
```

```
        // AccumulateInterest - начисление процентов
        public void AccumulateInterest()
        {
            _balance = Balance + (Balance * _interestRate);
        }

        // Withdraw - снятие со счета произвольной суммы, не
        // превышающей имеющейся на счету; возвращает снятую
        // сумму
        override public decimal Withdraw(decimal withdrawal)
        {
            Console.WriteLine("SavingsAccount.Withdraw()...");
            Console.WriteLine("Вызов метода Withdraw базового " +
                              "класса дважды...");
            // Снятие 1.50
            base.Withdraw(1.5M);
            // Снятие в пределах оставшейся суммы
            return base.Withdraw(withdrawal);
        }
    }

    public class Program
    {
        public static void MakeAWithdrawal(BankAccount ba,
                                           decimal amount)
        {
            ba.Withdraw(amount);
        }

        public static void Main(string[] args)
        {
            BankAccount ba;
            SavingsAccount sa;
            // Вывод баланса
            Console.WriteLine("MakeAWithdrawal(ba, ...)");
            ba = new BankAccount(200M);
            MakeAWithdrawal(ba, 100M);
            Console.WriteLine("Баланс BankAccount равен {0:C}",
                              ba.Balance);
            Console.WriteLine("MakeAWithdrawal(sa, ...)");
            sa = new SavingsAccount(200M, 12);
            MakeAWithdrawal(sa, 100M);
            Console.WriteLine("Баланс SavingsAccount равен {0:C}",
                              sa.Balance);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

Вывод программы имеет следующий вид:

```
MakeAWithdrawal(ba, ...)
BankAccount.Withdraw() с $100...
Баланс BankAccount равен $100.00
MakeAWithdrawal(sa, ...)
SavingsAccount.Withdraw()...
Вызов метода Withdraw базового класса дважды...
BankAccount.Withdraw() с $1.5...
BankAccount.Withdraw() с $100...
Баланс SavingsAccount равен $98.50
Нажмите <Enter> для завершения программы...
```


ЗАПОМНИ!

Метод `Withdraw()` помечен в базовом классе `BankAccount` как `virtual`, в то время как в подклассе он помечен как `override`. Метод `MakeAWithdrawal()` остается без изменений, и вывод при его вызове различен из-за того, что разрешение вызова `ba.Withdraw()` осуществляется на основании типа `ba` во время выполнения программы.


СОВЕТ

Будьте экономны при объявлении методов виртуальными. Все имеет свою цену, так что используйте ключевое слово `virtual` только при необходимости. Это компромисс между классом, который очень гибок и может быть перекрыт (со множеством виртуальных методов), и классом, который недостаточно гибок (и не содержит ничего виртуального).

Получение максимальной выгоды от полиморфизма

Мощь полиморфизма во многом основана на том, что полиморфные объекты используют один и тот же интерфейс. Например, в иерархии объектов `Shape` — `Circle`, `Square`, `Triangle` и др. — можно считать, что все виды фигур имеют свой метод `Draw()`. Метод `Draw()` каждого из объектов, само собой, реализован по-своему. Но главное в том, что для коллекции таких объектов можно использовать цикл `foreach`, который будет вызывать метод `Draw()` (или любой другой метод полиморфного интерфейса объектов).

Визитная карточка класса: метод `ToString()`

Все классы наследуют общий базовый класс с именем `Object`. Стоит упомянуть, что этот класс включает метод `ToString()`, который преобразует содержимое объекта в строку `string`. Идея в том, что каждый класс должен перекрывать метод `ToString()`, чтобы иметь возможность вывода своих объектов.

Ранее я использовал для этой цели метод `GetString()`, чтобы не смущать читателя до тех пор, пока он не познакомится с наследованием. Теперь же, когда вы познакомились с наследованием, ключевым словом `virtual` и перекрытием методов, можно поговорить и о методе `ToString()`. Перекрывая этот метод для каждого класса, вы позволяете им выводить свои объекты наиболее подходящим образом. Например, метод `Student.ToString()` может выводить имя студента и его идентификатор.

Большинство методов — даже встроенных в библиотеку C# — используют для вывода объектов этот метод. Таким образом, перекрытие метода `ToString()` имеет очень полезное побочное действие, заключающееся в том, что объект будет выводиться в собственном уникальном формате, независимо от того, как именно и кем он будет выводиться.

Абстракционизм в C#

Утка — вид птицы. Так же, как воробей или колибри. Любая птица представляет какой-то подвид птиц. Но обратная сторона медали в том, что *нет* птицы, которая была бы птицей вообще. С точки зрения программирования это означает, что все объекты `Bird` являются экземплярами каких-то подклассов `Bird`, но не существует ни одного экземпляра класса `Bird`. Так что же такое "птица"? Это всегда какой-то конкретный вид — пингвин, курица или, к примеру, страус.

Различные типы птиц имеют множество общих свойств (в противном случае они бы не были птицами), но нет двух типов, у которых бы общими были все свойства. Если бы такие типы были, они были бы одинаковыми типами, ничем не отличающимися друг от друга.

Разложение классов

Люди систематизируют объекты, выделяя их общие черты. Чтобы увидеть, как это работает, рассмотрим два класса — `HighSchool` и `University`, — показанные на рис. 17.1. Здесь для описания классов использован Унифицированный язык моделирования (Unified Modeling Language — UML), графический язык, описывающий классы и их взаимоотношения друг с другом.

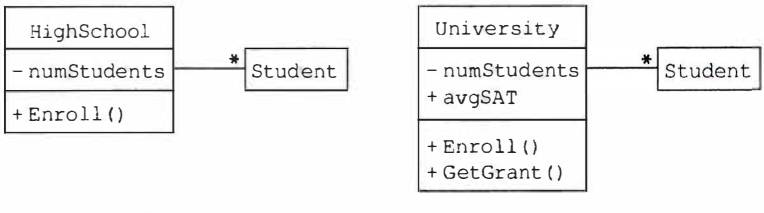


Рис. 17.1. UML-описание классов `HighSchool` и `University`

Как видно на рис. 17.1, у школы и университета много общих свойств. И у школы, и у университета имеется открытый метод `Enroll()` для добавления объекта `Student` (зачисления в учебное заведение). Оба класса имеют закрытый член `numStudents`, в котором хранится число учащихся. Еще одно общее свойство — взаимоотношения учащихся и учебных заведений: в учебном заведении может быть много учащихся, в то время как один учащийся учится одновременно только в одном учебном заведении. Само собой, имеется масса других свойств учебных заведений, но для данного рассмотрения ограничимся перечисленным.

В дополнение к свойствам школы университет содержит метод `GetGrant()` и член-данные `avgSAT`.

UML LITE

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Унифицированный язык моделирования (Unified Modeling Language — UML) представляет собой выразительный язык, способный ясно определять взаимоотношения объектов в программе. Одно из достоинств UML заключается в том, что вы можете не зависеть от конкретного языка программирования.

Ниже перечислены основные свойства UML.

- Классы представлены прямоугольниками, разделенными по вертикали на три части. Имя класса указывается в верхней части прямоугольника.
- Члены-данные класса находятся в средней части, а методы — в нижней. Можно опустить среднюю или нижнюю часть прямоугольника, если в классе нет членов-данных или методов.
- Члены со знаком плюс (+) перед именем являются открытыми, со знаком минус (–) — закрытыми. В UML отсутствует специальный знак для защищенных членов, но некоторые программисты используют для обозначения таких членов символ #. Закрытые члены доступны только для других членов того же класса; открытые члены доступны всем классам.
- Метка `{abstract}` после имени указывает абстрактный класс или метод. На самом деле UML использует для этого иное обозначение, но так мне кажется проще. Можно также использовать для абстрактных методов курсив.
- Стрелка между двумя классами представляет отношение между ними. Число над линией означает мощность — сколько элементов может быть с каждого конца стрелки. Звездочка (*) означает *произвольное число*. Если число опущено, по умолчанию предполагается значение 1. Таким образом, на рис. 17.1 видно, что один университет может иметь сколько угодно студентов — они связаны отношением "один-ко-многим".

- Линия с большой открытой или треугольной стрелкой на конце выражает отношение ЯВЛЯЕТСЯ (наследование). Стрелка указывает в иерархии классов на базовый класс. Другие типы взаимоотношений включают отношение СОДЕРЖИТ, которое указывается линией с закрашенным ромбиком со стороны владельца.

Схема на рис. 17.1 приемлема, насколько это возможно, но бо́льшая часть информации дублируется, а дублирование в коде (и диаграммах UML) оставляет неприятный душок. Вы можете уменьшить дублирование, позволяя наследовать более сложный класс `University` от более простого класса `HighSchool`, как показано на рис. 17.2.

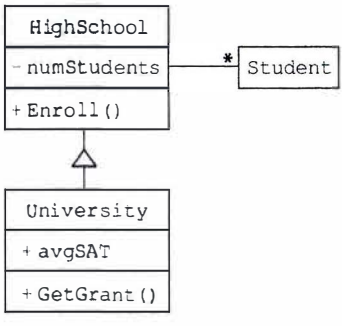


Рис. 17.2. Наследование `HighSchool` упрощает класс `University`, но приводит к проблемам

Класс `HighSchool` остается неизменным, но класс `University` при этом проще описать. Можно сказать, что `University` — это класс `HighSchool` с членом `avgSAT` и методом `GetGrant()`. Однако такое решение имеет одну фундаментальную проблему: университет — это вовсе не школа со специальными свойствами.

Вы можете сказать "Ну и что? Главное, что наследование работает и экономит наши усилия". Да, конечно, это так, но сказанное выше — не просто стилистическая тривиальность. Такое неверное представление может ввести в заблуждение программиста как сейчас, так и в будущем. В один прекрасный день ему, незнакомому с вашими фокусами, придется читать и разбираться в ваших исходных текстах, и такое неверное представление может привести к неправильному пониманию программы.

Кроме того, неверное представление может привести к реальным проблемам. Предположим, что в школе решили выбирать лучшего ученика, и для этого программист просто добавляет в класс `HighSchool` метод `NameFavorite()`, указывающий имя такого ученика.

И вот — проблема. В университете не намерены определять лучшего студента, но метод `NameFavorite()` оказывается унаследованным. Это может показаться небольшой проблемой — в конце концов, этот метод в классе `University` можно просто игнорировать. Да, один лишний метод не делает погоды, но это еще один кирпич в стене непонимания. Постепенно лишние члены-данные и методы накапливаются, и наступает момент, когда ваш класс уже не в состоянии вынести такой багаж. Несчастный программист уже не понимает, какие методы "реальны", а какие — нет.


ЗАПОМНИ!

Наследование для удобства приводит и к другим проблемам. При наследовании, показанном на рис. 17.2, как видно из схемы, классы `University` и `HighSchool` имеют одну и ту же процедуру зачисления. Как бы странно это ни звучало, будем считать, что это так и есть. Программа разработана, упакована и отправлена потребителям. Несколькими месяцами позже министерство просвещения решает изменить правила зачисления в школы, что, в свою очередь, приводит к изменению процедуры зачисления и в университеты, что, конечно же, неверно.

Чтобы избежать указанной проблемы, следует осознать, что университет не является разновидностью школы. Отношение СОДЕРЖИТ также не будет работать — ведь университет не содержит школу, как и школа не содержит университет. Решение заключается в том, что и школа, и университет — это специальные типы учебных заведений.

На рис. 17.3 показано более корректное отношение. Новый класс `School` содержит общие свойства двух типов учебных заведений, включая отношения с объектами `Student`. Более того, класс `School` даже имеет метод `Enroll()`, хотя он и абстрактный, поскольку и `University`, и `HighSchool` реализуют его по-разному.

Теперь классы `University` и `HighSchool` наследуют общий базовый класс. Каждый из них содержит свои уникальные члены: `HighSchool` — `NameFavorite()`, а `University` — `GetGrant()`. Кроме того, оба класса перекрывают метод `Enroll()`, описывающий правила зачисления учащихся в разные учебные заведения. По сути, здесь выделено общее путем создания базового класса из двух схожих классов, которые после этого стали подклассами. Введение класса `School` имеет как минимум два больших преимущества.

» **Это соответствует реальности.** Университет является учебным заведением, но не школой. Соответствие действительности — важное, но не главное преимущество.

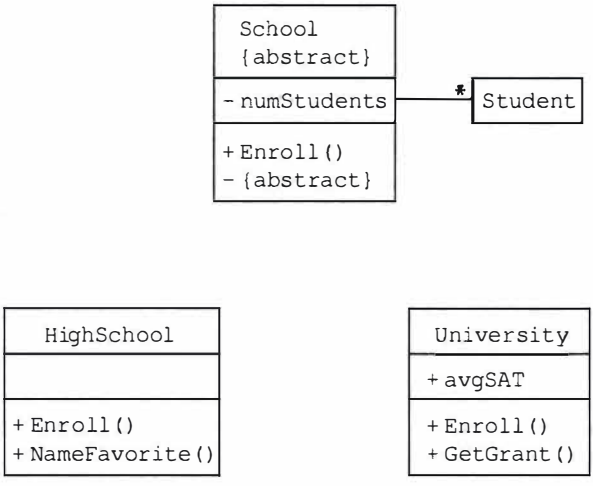


Рис. 17.3. Классы `University` и `HighSchool` должны иметь общий базовый класс `School`

» **Это изолирует один класс от изменений или дополнений в другой класс.** Если потребуется, например, внести дополнения в класс `University`, то его новые методы никак не повлияют на класс `HighSchool`.

Процесс выделения общих свойств из схожих классов называется *разложением* классов (factoring). Это важное свойство объектно-ориентированных языков программирования как по описанным выше причинам, так и с точки зрения снижения избыточности.


ВНИМАНИЕ!

Разложение корректно только в том случае, если отношения наследования соответствуют действительности. Можно выделять общие свойства классов `Mouse` и `Joystick`, поскольку оба они представляют собой указывающие устройства, но делать то же для классов `Mouse` и `Display` будет ошибкой.

Разложение обычно приводит к нескольким уровням абстракции. Например, программа, охватывающая более широкий круг школ, может иметь структуру классов, показанную на рис. 17.4.

Как видите, внесено два новых класса между `University` и `School`: `HigherLearning` и `LowerLevel`. Например, новый класс `HigherLearning` делится на классы `College` и `University`. Такая многослойная иерархия — обычное и даже желательное явление при разложении, соответствующем реальному миру.

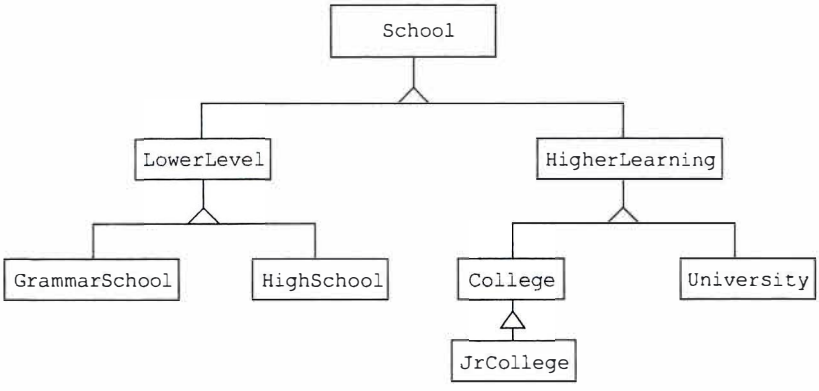


Рис. 17.4. Разложение классов обычно дает дополнительные уровни в иерархии наследования

Заметим, однако, что никакой Единой Теории Разложения не существует. Так, разложение на рис. 17.4 можно считать вполне естественным, но если программа в большей степени связана с вопросами администрирования учебных заведений местными властями, то еще более естественной будет иерархия классов, представленная на рис. 17.5.

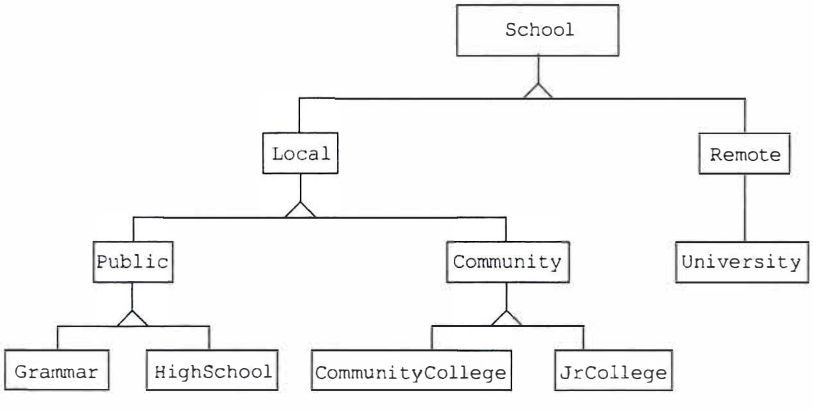


Рис. 17.5. Разложение классов зависит от решаемой задачи

Абстрактный класс: ничего, кроме идеи

Вернемся в очередной раз к классу `BankAccount`. Подумайте, как вы можете определить различные методы-члены, определенные в `BankAccount`.

Большинство методов этого класса не вызывают проблем, поскольку оба типа банковских счетов одинаково их реализуют. Однако правила снятия со счета оказываются различными, так что вы должны реализовать

SaveingsAccount.Withdraw() не так, как CheckingAccount.Withdraw(). Но как же реализовать исходный метод BankAccount.Withdraw()? Давайте обратимся за помощью к банковскому служащему. Представляете этот диалог?

— Каковы правила снятия денег со счета? — спрашиваете вы.

— С какого счета? Депозитного или чекового?

— Со счета, — отвечаете вы. — Просто со счета.

Полное непонимание в ответ.

Проблема в том, что заданный вопрос не имеет смысла. Не существует такой вещи, как "просто счет". Все счета (в анализируемом примере) являются либо депозитными, либо чековыми. Концепция счета представляет собой абстракцию, которая объединяет общие свойства конкретных счетов. Она оказывается неполной, поскольку в ней недостает важного свойства Withdraw() (если немного поразмышлять, то найдутся и другие отсутствующие свойства).

Как использовать абстрактные классы

Абстрактные классы используются для описания абстрактных концепций. *Абстрактный класс* — это класс с одним или несколькими абстрактными методами. Абстрактный метод — это метод, описанный при помощи ключевого слова abstract и не имеющий реализации. Тело такого метода создается тогда, когда вы порождаете подкласс абстрактного класса. Рассмотрим следующую (урезанную) демонстрационную программу:

```
// AbstractInheritance - класс BankAccount является
// абстрактным, поскольку в нем не существует реализации
// метода Withdraw()
using System;
namespace AbstractInheritance
{
    // AbstractBaseClass - создадим абстрактный класс, в
    // котором имеется только единственный метод Output()
    abstract public class AbstractBaseClass
    {
        // Output - абстрактный метод, который выводит строку,
        // но только в подклассах, которые перекрывают этот
        // метод
        abstract public void Output(string outputString);
    }

    // SubClass1 - первая конкретная реализация класса
    // AbstractBaseClass
    public class SubClass1 : AbstractBaseClass
    {
        override public void Output(string source)
        {
            string s = source.ToUpper();
            Console.WriteLine("Вызов SubClass1.Output() из {0}", s);
        }
    }
```

```
    // SubClass2 - еще одна конкретная реализация класса
    // AbstractBaseClass
    public class SubClass2 : AbstractBaseClass
    {
        override public void Output(string source)
        {
            string s = source.ToLower();
            Console.WriteLine("Вызов SubClass2.Output() из {0}",
                              s);
        }
    }

    class Program
    {
        public static void Test(AbstractBaseClass ba)
        {
            ba.Output("Test");
        }

        public static void Main(string[] strings)
        {
            // Нельзя создать объект класса AbstractBaseClass,
            // поскольку он - абстрактный. Если вы снимете
            // комментарий со следующей строки, то C# сгенерирует
            // сообщение об ошибке компиляции
            // AbstractBaseClass ba = new AbstractBaseClass();

            // Теперь повторим наш эксперимент с классом Subclass1
            Console.WriteLine("Создание объекта SubClass1");
            SubClass1 sc1 = new SubClass1();
            Test(sc1);

            // и классом Subclass2
            Console.WriteLine("Создание объекта SubClass2");
            SubClass2 sc2 = new SubClass2();
            Test(sc2);

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

В программе сначала определяется класс `AbstractBaseClass` с единственным абстрактным методом `Output()`. Поскольку он объявлен как `abstract`, метод `Output()` не имеет реализации, т.е. тела метода. Класс `AbstractBaseClass` наследуют два подкласса: `SubClass1` и `SubClass2`. Оба класса — конкретные, так как перекрывают метод `Output()` "настоящими" методами и не содержат собственных абстрактных методов.

СОВЕТ

Класс может быть объявлен как абстрактный независимо от наличия в нем абстрактных методов. Однако конкретным класс может быть тогда и только тогда, когда все абстрактные методы всех базовых классов выше него перекрыты реальными методами.

Методы `Output()` двух рассматриваемых подклассов немного различны: один из них преобразует передаваемую ему строку в верхний регистр, другой — в нижний. Вывод программы демонстрирует полиморфную природу класса `AbstractBaseClass`.

```
Создание объекта SubClass1
Вызов SubClass1.Output() из TEST
Создание объекта SubClass2
Вызов SubClass2.Output() из test
Нажмите <Enter> для завершения программы...
```

СОВЕТ

Абстрактный метод автоматически является виртуальным, так что добавлять ключевое слово `virtual` к ключевому слову `abstract` не требуется.

Создание абстрактных объектов невозможно

Обратите внимание еще на одну вещь в рассматриваемой демонстрационной программе: нельзя создавать объект `AbstractBaseClass`, но аргумент метода `Test()` объявлен как объект класса `AbstractBaseClass` *или одного из его подклассов*. Это дополнение крайне важно. Объекты `SubClass1` и `SubClass2` могут быть переданы в метод, поскольку оба являются конкретными подклассами `AbstractBaseClass`. Здесь использовано отношение ЯВЛЯЕТСЯ. Это очень мощная методика, позволяющая писать высокообобщенные методы.

Опечатывание класса

Вы можете решить, что последующие поколения программистов недостойны расширять написанный вами класс. Заблокировать его от возможных расширений можно посредством ключевого слова `sealed` — такой класс не сможет выступать в качестве базового. Рассмотрим следующий фрагмент исходного текста:

```
using System;
public class BankAccount
{
    // Withdrawal - вы можете снять со счета любую сумму, не
    // превышающую баланс. Возвращает реально снятую со
```

```
    // счета сумму
    virtual public void Withdraw(decimal withdraw)
    {
        Console.WriteLine("Вызов BankAccount.Withdraw()");
    }
}

public sealed class SavingsAccount : BankAccount
{
    override public void Withdraw(decimal withdrawal)
    {
        Console.WriteLine("Вызов SavingsAccount.Withdraw()");
    }
}

public class SpecialSaleAccount : SavingsAccount
{
    override public void Withdraw(decimal withdrawal)
    {
        Console.WriteLine("Вызов " +
                          "SpecialSaleAccount.Withdraw()");
    }
}
```

При компиляции данного исходного текста вы получите следующее сообщение об ошибке:

```
'SpecialSaleAccount' : cannot inherit from sealed class 'SavingsAccount'
```

Ключевое слово `sealed` дает возможность защитить класс от вмешательства методов некоторых подклассов. Например, позволяя программисту расширять класс, реализующий систему безопасности, вы, по сути, разрешаете создать черный ход, минующий эту систему.

Опечатывание класса защищает другие программы, возможно, находящиеся где-то в Интернете, от применения модифицированной версии вашего класса. Удаленная программа может использовать класс таким, каков он есть, или не использовать вообще, но не может наследовать его с тем, чтобы использовать только его часть, перекрыв прочие методы.

Глава 18
Интерфейсы

В ЭТОЙ ГЛАВЕ...

- За кулисами ЯВЛЯЕТСЯ и СОДЕРЖИТ
- Создание и использование интерфейсов
- Унификация иерархий классов с помощью интерфейсов
- Повышение гибкости путем применения интерфейсов

Класс может *содержать* ссылку на другой класс. Это простое отношение СОДЕРЖИТ. Один класс может *расширять* другой класс с помощью наследования. Это — отношение ЯВЛЯЕТСЯ. Интерфейсы C# реализуют еще одно, не менее важное, отношение — МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК. В этой главе рассматривается, что же такое *интерфейсы* C#, и показаны несколько из множества способов повышения мощи и гибкости объектно-ориентированного программирования.

Что значит МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК

Если вы хотите написать памятку, можно взять ручку и бумагу либо воспользоваться органайзером или своим компьютером. Все эти объекты реализуют операцию "написать памятку" — `TakeANote`. Используя магию наследования, на языке C# это можно реализовать следующим образом:

```
abstract class ThingsThatRecord
{
    abstract public void TakeANote(string sNote);
}
public class Pen : ThingsThatRecord
{
    override public void TakeANote(string sNote)
    {
        // ... Написание заметки ручкой ...
    }
}
public class PDA : ThingsThatRecord
{
    override public void TakeANote(string sNote)
    {
        // ... при помощи органайзера ...
    }
}
public class Laptop : ThingsThatRecord
{
    override public void TakeANote(string sNote)
    {
        // ... еще каким-то образом ...
    }
}
```

СОВЕТ

Если ключевое слово `abstract` вас смущает, обратитесь за пояснениями к главе 17, "Полиморфизм". Если вам непонятно, что такое наследование, перечитайте главу 16, "Наследование". Следующий простой метод показывает, что подход с наследованием вполне работоспособен:

```
// Тип параметра функции представляет собой базовый класс
void RecordTask(ThingsThatRecord things)
{
    // Этот абстрактный метод реализован во всех классах,
    // которые наследуют ThingsThatRecord
    things.TakeANote("Список покупок");
    // ... и так далее ...
}
```

Типом параметра является `ThingsThatRecord`, так что вы можете передавать этому методу любые подклассы, что делает метод достаточно общим. Это может показаться хорошим решением, но у него есть два больших недостатка.

- **Первая проблема — фундаментальная**. Дело в том, что реально связать ручку, органайзер и компьютер соотношением ЯВЛЯЕТСЯ некорректно. Знание того, как работает ручка, не дает никаких сведений о том, как записывает информацию компьютер или органайзер.

» **Вторая проблема чисто техническая**. Гораздо лучше описать `Laptop` как подкласс класса `Computer`. Хотя `PDA` также можно наследовать от того же класса `Computer`, этого нельзя сказать о классе `Pen`. Вы можете охарактеризовать ручку как некоторый тип `MechanicalWriteDevice` (механическое пишущее устройство) или `DeviceThatStainsYourShirt` (устройство, пачкающее вашу рубашку). Однако в C# класс не может быть наследован от двух разных классов одновременно — класс C# может быть вещью только одного сорта.

Таким образом, классы `Pen`, `PDA` и `LapTop` имеют только одну общую характеристику: каждый из них МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК записывающее устройство. Наследование здесь неприменимо.

Что такое интерфейс

Интерфейс в C# выглядит очень похожим на класс без членов-данных, в котором все методы абстрактны, — очень похожим на абстрактный класс:

```
interface IRecordable
{
    void TakeANote(string note);
}
```

Обратите внимание на ключевое слово `interface` там, где обычно стоит ключевое слово `class`. В фигурных скобках интерфейса приведен список абстрактных методов. Интерфейсы не содержат определений каких-либо членов-данных или реализации методов. Однако интерфейсы могут содержать некоторые другие вещи, включая свойства (глава 15, "Класс: каждый сам за себя"), события (глава 19, "Делегирование событий") и индексаторы (глава 7, "Работа с коллекциями"). Список того, чего *не* может быть в интерфейсе C#, включает

- спецификаторы доступа, такие как `public` или `private` (глава 15, "Класс: каждый сам за себя");
- такие ключевые слова, как `virtual`, `override` или `abstract` (глава 17, "Полиморфизм");
- данные-члены (глава 12, "Немного о классах");
- реализованные методы — не абстрактные методы с телами.

Все члены интерфейса C# открыты (более того, вы даже не имеете права использовать какие-либо спецификаторы доступа при определении методов

интерфейса); интерфейс C# не может участвовать в обычном наследовании. (Сам интерфейс может быть указан как `public`, `protected`, `internal` или `private`.)

В отличие от абстрактного класса интерфейс C# классом не является. От него не могут быть порождены производные классы, а его методы не могут иметь тел.

Реализация интерфейса

Чтобы использовать интерфейс C#, вы должны *реализовать* его в одном или нескольких классах. Заголовок класса при этом имеет вид наподобие приведенного:

```
class Pen : IRecordable // Похоже на наследование, но
                        // наследованием не является
struct PenDescription : IRecordable
```

ЗАПОМНИ!

Интерфейс C# указывает элементы, для которых классы, реализующие интерфейс, должны определить конкретную реализацию. Например, любой класс, который реализует интерфейс `IRecordable`, должен предоставить реализацию метода `TakeANote`. Метод, который реализует `TakeANote`, не использует ключевое слово `override`. Эта реализация — не то же самое, что и перекрытие виртуального метода в классах. Класс `Pen` может выглядеть следующим образом:

```
class Pen : IRecordable
{
    // Реализация метода интерфейса
    public void TakeANote(string note)
    {   // ДОЛЖЕН быть объявлен как public
        // ... запись заметки ручкой ...
    }
}
```

Этот код удовлетворяет двум требованиям: указывает, что класс реализует интерфейс `IRecordable`, и предоставляет реализацию метода `TakeANote()`.

Синтаксис, указывающий, что класс наследует базовый класс, такой как `ThingsThatRecord`, по сути, не отличается от синтаксиса, указывающего, что класс реализует интерфейс C#, такой как `IRecordable`:

```
public class PDA : ThingsThatRecord ...
public class PDA : IRecordable ...
```

СОВЕТ

Интегрированная среда Visual Studio может помочь вам в реализации интерфейса. Разместите указатель мыши над именем интерфейса в заголовке класса. Под первым символом имени интерфейса появится небольшое подчеркивание. Перемещайте мышь, пока не откроется

меню, и выберите в нем пункт Implement interface <*name*>. При этом образуется заготовка реализации — вам надо только добавить в нее содержимое.

Именование интерфейсов

По соглашению об именованиях .NET имена интерфейсов начинаются с буквы `I`. Кроме того, для них, как правило, используются прилагательные, такие как `IRecordable`.

Зачем C# включает интерфейсы

Интерфейс описывает возможности и свойства, как, например, водительские права описывают умение водить автомобиль. Класс реализует интерфейс `IRecordable`, если он содержит полную версию метода `TakeANote`.

Кроме того, интерфейс представляет собой *контракт*. Если вы согласны реализовать все методы, определенные в интерфейсе, вы получите все его возможности. Не только их, конечно, но главное, что клиент, который будет использовать ваш класс, может вызывать методы интерфейса в полной уверенности, что они реализованы. Реализация интерфейса представляет собой обещание реализации методов, подкрепленное компилятором (обычно, если что-то гарантировано компилятором, это приводит к уменьшению количества ошибок).

Наследование и реализация интерфейса

В отличие от некоторых языков программирования наподобие C++, C# не допускает *множественное наследование* — наследование класса от двух и более базовых классов. Можно представить себе класс `HouseBoat`, унаследованный от классов `House` и `Boat`, но только не в C#.

Хотя класс и может наследовать только один базовый класс, *в дополнение* к этому он может реализовывать любое количество интерфейсов. После определения `IRecordable` в качестве интерфейса наши классы могут иметь следующий вид:

```
public class Pen : IRecordable // Базовый класс — Object
{
    public void TakeANote(string note)
    {
        // Запись ручкой
    }
}
```

```
public class PDA : ElectronicDevice, IRecordable
{
    public void TakeANote(string note)
    {
        // Запись в органайзер
    }
}
```

Класс `PDA` наследует базовый класс *и* реализует интерфейс.

Преимущества интерфейсов

Чтобы понять полезность интерфейса наподобие `IRecordable`, рассмотрим следующий код:

```
public class Program
{
    static public void RecordShoppingList(IRecordable recorder)
    {
        // Запись с использованием некоторого переданного
        // в качестве аргумента устройства
        recorder.TakeANote(...);
    }

    public static void Main(string[] args)
    {
        PDA pda = new PDA();
        RecordShoppingList(pda); // Аккумулятор сел...
        Pen pen = new Pen();
        RecordShoppingList(pen);
    }
}
```

Что означает параметр `IRecordable`? Это экземпляр любого класса, который реализует интерфейс `IRecordable`. Метод `RecordShoppingList()` не делает никаких предположений о точном типе записывающего объекта. Неважно, будет ли это органайзер, компьютер или карандаш, — лишь бы это устройство могло выполнить запись.

Это очень мощная концепция, поскольку позволяет методу `RecordShoppingList()` быть весьма обобщенным методом, что дает возможность широко применять его в других программах. Он даже более обобщенный, чем использование базового класса типа `ElectronicDevice` в качестве типа аргумента, поскольку интерфейс позволяет передать практически любой объект, который не обязательно имеет что-то общее с другими допустимыми объектами, помимо реализации интерфейса. Он даже не должен располагаться в той же иерархии классов. И это действительно упрощает разработку иерархий классов.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Перегруженный термин. Программисты используют термин *интерфейс* во многих случаях. Вы познакомились с ключевым словом C# `interface` и его применением. Говорят также об *открытом интерфейсе*, имея в виду открытые методы и свойства, доступные внешнему миру. Я постараюсь, чтобы читатель мог точно понимать, о чем идет речь, и в основном буду говорить об "интерфейсе C#", а когда речь будет идти о множестве отрытых методов класса, я буду говорить об "открытом интерфейсе".

Использование интерфейсов

Интерфейсы C# могут использоваться в качестве не только типов параметров, но и

- типа, возвращаемого методом;
- базового типа для высокообобщенного массива или коллекции;
- более общего вида ссылки на объект для типов переменных.

Преимущества применения интерфейсов C# в качестве типа параметра метода вы уже видели в предыдущем разделе. Давайте познакомимся с преимуществами и для других вариантов использования.

Тип, возвращаемый методом

В своих программах я предпочитаю делегировать задачу создания ключевых объектов *фабричному методу* (factory method). Предположим, что у меня есть такая переменная:

```
IRecordable recorder = null; // Переменная интерфейсного типа
```

Где-то, возможно в конструкторе, я вызываю фабрику для получения некоторой конкретной разновидности объекта `IRecordable`:

```
recorder = MyClass.CreateRecorder("Pen"); Метод-фабрика
                        // часто является статическим
```

Здесь `CreateRecorder()` представляет собой метод, зачастую того же класса, который возвращает не ссылку на `Pen`, а ссылку на `IRecordable`:

```
static IRecordable CreateRecorder(string recorderType)
{
    if(recorderType == "Pen") return new Pen();
    ...
}
```

Вы можете найти больше информации о фабриках в разделе "Что скрыто за интерфейсом" далее в этой главе. Обратите внимание, что тип возвращаемого значения `CreateRecorder()` является типом интерфейса.

Базовый тип массива или коллекции

Предположим, что у вас есть два класса, `Animal` и `Robot`. Как описать массив, который мог бы хранить как объект `thisCat` (типа `Animal`), так и `thatRobot` (типа `Robot`)? Единственный способ заключается в объявлении массива как хранящего объекты типа `Object`, первичного базового класса C#, и единственного базового класса, общего и для `Animal`, и для `Robot`:

```
object[] things = new object[] { thisCat, thatRobot };
```

Это плохое решение по множеству причин. Но предположим, что нас интересует передвижение объектов. В таком случае каждый класс может реализовывать интерфейс `IMovable`:

```
interface IMovable
{
    void Move(int direction, int speed, int distance);
}
```

Тогда массив может быть объявлен как содержащий элементы типа `IMovable`, что позволит работать с иначе несовместимыми объектами:

```
IMovable[] movables = { thisCat, thatRobot };
```

Интерфейс предоставляет вам общность классов, которую можно использовать в коллекциях.

Более общий тип ссылки

Приведенное далее объявление переменной ссылается на очень *конкретный* объект (см. раздел "Абстрактный или конкретный? Когда следует использовать абстрактный класс, а когда — интерфейс" данной главы):

```
Cat thisCat = new Cat();
```

Альтернативой является использование вместо ссылки интерфейса C#:

```
IMovable thisMovableCat = (IMovable)new Cat(); // Обратите
              // внимание на необходимость приведения типа
```

Теперь данной переменной можно присваивать любой объект, который реализует интерфейс `IMovable`. Этот метод широко применяется в объектно-ориентированном программировании, как вы увидите ниже в данной главе.

Использование предопределенных типов интерфейсов C#

Поскольку интерфейсы оказываются столь полезными, в библиотеке классов .NET можно обнаружить огромное их количество. В справочной системе я насчитал их несколько десятков, после чего сбился со счета и бросил это занятие. Среди интерфейсов в пространстве имен `System` можно упомянуть такие, как `IComparable`, `IComparable<T>`, `IDisposable` и `IFormattable`. Пространство имен `System.Collections.Generics` включает такие интерфейсы, как `IEnumerable<T>`, `IList<T>`, `ICollection<T>` и `IDictionary<TKey,TValue>`. Имеются и многие другие. Интерфейсы с записью `<T>` представляют собой обобщенные интерфейсы. Что означает `<T>`, я пояснял в главе 6, "Глава для коллекционеров", при рассмотрении классов коллекций.

Два очень часто использовавшихся интерфейса — `IComparable` и `IEnumerable` — в настоящее время заменены их обобщенными версиями `IComparable<T>` и `IEnumerable<T>`.

Ниже в этой главе рассмотрен интерфейс `IComparable<T>`, который делает возможным сравнение всех видов объектов (таких, например, как `Student`) один с другим, так что можно использовать метод `Sort()`, применимый ко всем массивам и большинству коллекций. Интерфейс `IEnumerable<T>` делает возможной работу цикла `foreach`. Большинство коллекций реализуют этот интерфейс, так что их можно итерировать с применением цикла `foreach`. Еще одно применение интерфейса `IEnumerable<T>` — в качестве основы для выражений запросов в C# 3.0 и более поздних версиях языка.

Пример программы, использующей отношение МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК

Интерфейсы помогают выполнять такие задачи, на которые не способны классы, потому что вы можете реализовать столько интерфейсов, сколько захотите, но наследовать можно только один класс. Рассматриваемая далее программа `SortInterface` демонстрирует эффективное применение множественных интерфейсов на практике.

Создание собственного интерфейса

Интерфейс `IDisplayable` удовлетворяется любым классом, который содержит метод `Display()` (и, само собой, объявляет, что он реализует

IDisplayable). Display() возвращает объект типа string, который может быть выведен на экран с использованием WriteLine():

```
// IDisplayable - объект, реализующий метод Display()
interface IDisplayable
{
    // Возвращает собственное описание
    string Display();
}
```

Приведенный далее класс Student реализует интерфейс IDisplayable:

```
class Student : IDisplayable
{
    public Student(string name, double grade)
    {
        Name = name;
        Grade = grade;
    }
    public string Name { get; private set; }
    public double Grade { get; private set; }
    public string Display()
    {
        string padName = Name.PadRight(9);
        return String.Format("{0}: {1:N0}", padName, Grade);
    }
}
```

Вызов метода PadRight() класса String делает вывод на экран более привлекательным.

Приведенный далее метод DisplayArray() получает массив любых объектов, которые реализуют интерфейс IDisplayable. Каждый из этих объектов гарантированно (гарантия обеспечивается интерфейсом) имеет собственный метод Display(). (Полностью программа будет приведена ниже.)

```
// DisplayArray - вывод массива объектов, которые
// реализуют интерфейс IDisplayable
public static void DisplayArray(IDisplayable[] displayables)
{
    foreach(IDisplayable disp in displayables)
    {
        Console.WriteLine("{0}, disp.Display());
    }
}
```

Вот пример вывода данного метода:

```
Homer    : 0
Marge    : 85
Bart     : 50
Lisa     : 100
Maggie   : 30
```

Реализация интерфейса IComparable<T>

Язык C# определяет интерфейс IComparable<T> следующим образом:

```
interface IComparable<T>
{
    // Сравнивает текущий объект типа T с объектом 'item';
    // возвращает 1, если текущий объект больше, -1, если
    // меньше, и 0 в противном случае
    int CompareTo(T item);
}
```

Класс реализует интерфейс IComparable<T> путем реализации метода CompareTo(). Обратите внимание на то, что метод CompareTo() получает аргумент некоторого типа T, который вы определяете при *инстанцировании интерфейса* для конкретного типа данных, как в следующем примере:

```
class SoAndSo : IComparable<SoAndSo> // Возможность сравнения
```

При реализации интерфейса IComparable<T> для вашего класса его метод CompareTo() должен возвращать 0, если сравниваемые элементы (вашего типа) "одинаковы" при некотором определяемом вами способе сравнения. Если нет, метод должен возвращать 1 или –1, в зависимости от того, какой из элементов "больше".

Как ни странно, но отношение сравнения можно задать и для объектов типа Student, например по их успеваемости. Реализация метода CompareTo() приводит к тому, что объекты могут быть отсортированы. Если один студент "больше" другого, их можно расположить от "меньшего" к "большему". На самом деле в большинстве классов коллекций (включая массивы, но не словари) уже реализован метод Sort():

```
void Sort(IComparable<T>[] objects);
```

Этот метод сортирует массив объектов, которые реализуют интерфейс IComparable<T>. Не имеет значения, к какому классу в действительности принадлежат объекты, например это могут быть объекты Student. Классы коллекций наподобие массивов или List<T> могут сортировать следующую версию Student:

```
// Student - описание студента с использованием имени и
// успеваемости
class Student : IComparable<Student>, IDisplayable
{
    // Конструктор – инициализация нового объекта
    public Student(double grade)
    {
        Grade = grade;
    }
    public double Grade { get; private set; }
```

```
    // Реализация интерфейса IComparable<T>:
    // CompareTo - сравнение двух студентов; студент с лучшей
    // успеваемостью "больше"
    public int CompareTo(Student rightStudent)
    {
        Student leftStudent = this;
        if (rightStudent.Grade < leftStudent.Grade)
        {
            return -1;
        }
        if (rightStudent.Grade > leftStudent.Grade)
        {
            return 1;
        }
        return 0;
    }
}
```

Сортировка массива объектов `Student` сводится к единственному вызову:

```
// Student реализует IComparable<T>
void MyMethod(Student[] students)
{
    Array.Sort(students); // Сортировка массива IComparable<Student>
}
```

Ваше дело — обеспечить компаратор (`CompareTo()`); `Array` сделает все остальное сам.

Сборка воедино

И вот наступил долгожданный момент: полная программа `SortInterface`, использующая описанные ранее возможности, готова.

```
// SortInterface - демонстрационная программа SortInterface
// иллюстрирует концепцию интерфейса
using System;
namespace SortInterface
{
    // IDisplayable - объект, который может представить
    // информацию о себе в строковом формате
    interface IDisplayable
    {
        // Display - возврат строки, предоставляющей
        // информацию об объекте
        string Display();
    }
    class Program
    {
        public static void Main(string[] args)
        {
            // Сортировка студентов по успеваемости...
            Console.WriteLine("Сортировка списка студентов");
```

```
        // Получаем неотсортированный список студентов
        Student[] students = Student.CreateStudentList();

        // Используем интерфейс IComparable<T> для сортировки
        // массива
        Array.Sort(students);

        // Теперь интерфейс IDisplayable выводит результат
        DisplayArray(students);

        // Теперь отсортируем массив птиц по имени с
        // использованием той же процедуры, хотя классы Bird
        // и Student не имеют общего базового класса
        Console.WriteLine("\nСортировка списка птиц");
        Bird[] birds = Bird.CreateBirdList();

        // Обратите внимание на отсутствие необходимости
        // явного преобразования типа объектов...
        Array.Sort(birds);
        DisplayArray(birds);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }

    // DisplayArray - вывод массива объектов, реализующих
    // интерфейс IDisplayable
    public static void
    DisplayArray(IDisplayable[] displayables)
    {
        foreach (IDisplayable displayable in displayables)
        {
            Console.WriteLine("{0}", displayable.Display());
        }
    }
}

// ----------- Students - сортировка по успеваемости ----
// Student - описание студента с использованием имени и
// успеваемости
class Student : IComparable<Student>, IDisplayable
{
    // Конструктор - инициализация нового объекта
    public Student(string name, double grade)
    {
        Name = Name;
        Grade = grade;
    }

    // CreateStudentList - для простоты создаем
    // фиксированный список студентов
```

```
    static string[] names = {"Homer", "Marge", "Bart",
                             "Lisa", "Maggie"
                            };
    static double[] grades = {0, 85, 50, 100, 30};
    public static Student[] CreateStudentList()
    {
        Student[] students = new Student[names.Length];
        for (int i = 0; i < names.Length; i++)
        {
            students[i] = new Student(names[i], grades[i]);
        }
        return students;
    }

    // Методы доступа только для чтения
    public string Name { get; private set; }
    public double Grade { get; private set; }

    // Реализация интерфейса IComparable:
    // CompareTo - сравнение двух объектов (в нашем случае -
    // объектов типа Student) и выяснение, какой из
    // них должен идти раньше в отсортированном списке
    public int CompareTo(Student rightStudent)
    {
        // Сравнение текущего Student (назовем его левым) и
        // другого (назовем его правым)
        Student leftStudent = this;

        // Генерируем -1, 0 или 1 на основании критерия
        // сортировки
        if (rightStudent.Grade < leftStudent.Grade)
        {
            return -1;
        }
        if (rightStudent.Grade > leftStudent.Grade)
        {
            return 1;
        }
        return 0;
    }

    // Реализация интерфейса IDisplayable:
    public string Display()
    {
        string padName = Name.PadRight(9);
        return String.Format("{0}: {1:N0}", padName, Grade);
    }
}

// ----------Birds - сортировка птиц по именам--------
// Массив имен птиц
class Bird : IComparable<Bird>, IDisplayable
{
```

```
        // Конструктор - инициализация объекта Bird
        public Bird(string name)
        {
            Name = name;
        }

        // CreateBirdList - возвращает список птиц; для простоты
        // используем фиксированный список
        static string[] birdNames =
        {
            "Oriole", "Hawk", "Robin", "Cardinal",
            "Bluejay", "Finch", "Sparrow"
        };
        public static Bird[] CreateBirdList()
        {
            Bird[] birds = new Bird[birdNames.Length];

            for (int i = 0; i < birds.Length; i++)
            {
                birds[i] = new Bird(birdNames[i]);
            }

            return birds;
        }

        public string Name { get; private set; }

        // Реализация интерфейса IComparable:
        // CompareTo - сравнение имен птиц; используется
        // встроенный метод сравнения класса String
        public int CompareTo(Bird rightBird)
        {
            // Сравнение текущего Bird (назовем его левым) и
            // другого (назовем его правым)
            Bird leftBird = this;
            return String.Compare(leftBird.Name, rightBird.Name);
        }

        // Реализация интерфейса IDisplayable:
        // Display - возвращает строку с именем птицы
        public string Display()
        {
            return Name;
        }
    }
}
```

Класс `Student` (примерно в середине листинга) реализует интерфейсы `IComparable<T>` и `IDisplayable`, как описано ранее. Метод `CompareTo()` сравнивает студентов по успеваемости, что приводит к соответствующей сортировке их списка. Метод `Display()` возвращает имя и успеваемость студента.

Прочие методы класса `Student` включают свойства только для чтения `Name` и `Grade`, простой конструктор и метод `CreateStudentList()`. Последний метод просто возвращает фиксированный список студентов.

Класс `Bird` внизу листинга также реализует интерфейсы `IComparable<T>` и `IDisplayable`. Он реализует метод `CompareTo()`, который сравнивает названия птиц посредством метода `String.Compare()`. Таким образом, в результате сортировки получается список птиц в алфавитном порядке. Метод `Display()` просто возвращает название птицы.

Вернемся к `Main()`

Теперь можно вернуться к функции `Main()`. Метод `CreateStudentList()` используется для получения неотсортированного списка, который сохраняется в массиве `students`. Можно решить, что для передачи массива студентов методу `Array.Sort()` необходимо приведение типа массива студентов к массиву элементов типа `comparableObjects`:

```
IComparable<Student>[] comparables =
            (IComparable<Student>[])students;
```

Но на самом деле это не так. Метод `Sort()` видит, что переданный ему массив содержит объекты, реализующие `IComparable<something>`, и просто вызывает метод `CompareTo()` для каждого объекта `Student` для их сортировки. Затем отсортированный массив объектов типа `Student` передается локально определенному методу `DisplayArray()`, итеративно проходящему по всем элементам массива объектов, которые гарантированно реализуют метод `Display()`, так как они реализуют интерфейс `IDisplayable`. Для каждого объекта вызывается метод `Display()`, и его результат выводится на дисплей с помощью метода `WriteLine()`.

Далее программа сортирует и выводит список птиц. Несомненно, вы согласитесь, что между птицами и студентами нет ничего общего. Однако класс `Bird` реализует интерфейс `IComparable` путем сравнения названий птиц и интерфейс `IDisplayable` путем возврата названий птиц.

```
Сортировка списка студентов
Lisa     : 100
Marge    : 85
Bart     : 50
Maggie   : 30
Homer    : 0

Сортировка списка птиц
Blue jay
Cardinal
Finch
Hawk
```

```
Oriole
Robin
Sparrow
Нажмите <Enter> для завершения программы...
```

Унификация иерархий классов

На рис. 18.1 показаны иерархии `Robot` и `Animal`. Некоторые, но не все классы в каждой иерархии не только наследуют базовые классы `Robot` или `Animal`, но и реализуют интерфейс `IPet` (не все животные являются домашними).

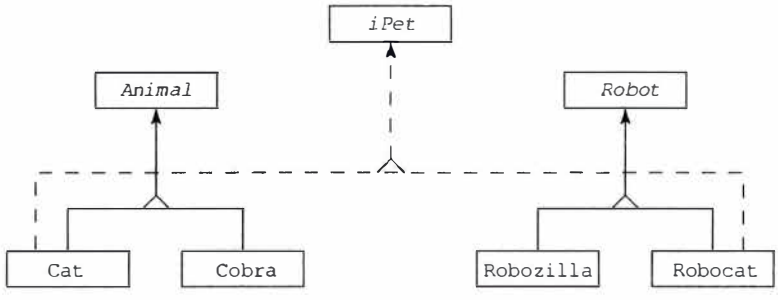


Рис. 18.1. Две иерархии классов и один интерфейс

В приведенном далее коде показана реализация этой иерархии. Обратите внимание на свойства в `IPet` — каким образом можно указывать свойства в интерфейсах. Если вам нужны оба метода — и чтения, и записи, — просто добавьте `set;` после `get;`.

```
// Два абстрактных базовых класса и один интерфейс
abstract class Animal
{
    abstract public void Eat(string food);
    abstract public void Sleep(int hours);
    abstract public int NumberOfLegs { get; }
    public void Breathe()
    {           // Не абстрактный,
        ...     // реализация имеется, но не
    }           // показана для краткости
}

abstract class Robot
{
    public virtual void Speak(string whatToSay)
    {           // Не абстрактный,
        ...     // реализация имеется, но не
    }           // показана для краткости
    abstract public void LiftObject(object o);
    abstract public int NumberOfLegs { get; }
```

```
}

interface IPet
{
    void AskForStrokes();
    void DoTricks();
    int NumberOfLegs { get; } // Свойства в интерфейсах должны
                              // иметь подобный вид.
    string Name { get; set; } // get/set в реализациях должны
                              // быть открытыми
}

// Cat - конкретный класс, который наследует (и частично
// реализует) класс Animal, а также реализует интерфейс IPet.
class Cat : Animal, IPet
{
    public Cat(string name)
    {
        Name = name;
    }
    // 1. Перекрывает и реализует члены Animal (не показаны).
    // 2. Представляет дополнительную реализацию IPet.
#region IPet Members
    public void AskForStrokes() ...
    public void DoTricks() ...
    public string Name { get; set; }
    // Наследует свойство NumberOfLegs от базового класса, тем
    // самым отвечая требованию IPet о наличии свойства
    // NumberOfLegs.
#endregion IPet Members
    public override string ToString()
    {
        return Name;
    }
}

class Cobra : Animal
{
    // 1. Перекрывает и реализует только все методы Animal
    // (не показаны).
}

class Robozilla : Robot // Не IPet.
{
    // 1. Перекрывает Speak.
    public override void Speak(string whatToSay)
    {
        Console.WriteLine("УНИЧТОЖИТЬ ВСЕХ ЛЮДЕЙ!");
    }
    // 2. Реализует LiftObject и NumberOfLegs (показано не все)
    public override void LiftObject(object o) ...
    public override int NumberOfLegs { get { return 2; } }
}
```

```
class RoboCat : Robot, IPet
{
    public RoboCat(string name)
    {
        Name = name;
    }
    // 1. Перекрывает некоторые члены Robot (показано не все)
#region IPet Members
    public void AskForStrokes() ...
    public void DoTricks() ...
    public string Name { get; set; }
#endregion IPet Members
}
```

В коде показаны два конкретных класса, которые являются наследниками `Animal`, и два, которые наследуют `Robot`. Однако ни класс `Cobra`, ни класс `Robozilla` не реализуют интерфейс `IPet` — вероятно, по уважительным причинам. Мне что-то не очень хочется по вечерам смотреть телевизор в компании кобры и Терминатора... Одни из классов в обеих иерархиях демонстрируют то, что можно назвать "одомашненностью", другие же этим свойством не обладают.

Главный вывод из этого раздела в том, что *любой* класс может реализовывать интерфейс, если он в состоянии обеспечить соответствующие методы и свойства. `Robotcat` и `Robodog` могут выполнять действия `AskForStrokes()` и `DoTricks()` и иметь свойство `NumberOfLegs` так же, как и `Cat` и `Dog` в иерархии `Animal`. Все прочие классы в тех же иерархиях не реализуют интерфейс `IPet`.

Что скрыто за интерфейсом

Зачастую в книге я рассматриваю код, который а) написан вами, но б) который кто-то другой (клиент) использует его в своих программах (конечно, вы можете быть своим собственным клиентом). Допустим, вы имеете столь сложный или хитроумный класс, что не хотите полностью предоставлять его открытый интерфейс клиентам, например он включает некоторые опасные операции, которые не должны быть доступны всем подряд. В идеале вы хотите предоставлять всем небольшое безопасное подмножество открытых методов, скрывая при этом другие, более опасные. Интерфейсы C# позволяют решить и эту задачу.

Вот еще один класс `Robozilla`, некоторые методы и свойства которого могут безопасно использоваться даже новичками от программирования. Но у класса есть и несколько более "продвинутых" возможностей, которые могут оказаться опасными в неумелых руках.

```
public class Robozilla // Не реализует IPet!
{
    public void ClimbStairs();                    // Безопасно
    public void PetTheRobodog();                  // Не очень опасно
    public void Charge();                         // Может быть опасно
    public void SearchAndDestroy();               // Опасно!
    public void LaunchGlobalThermonuclearWar(); // Беда...
}
```

Вы хотели бы предоставить всем желающим только два более безопасных метода, скрыв при этом три более опасных. Вот как это делается при помощи интерфейсов C#.

1. **Сначала разрабатываем интерфейс C#, который предоставляет безопасные методы:**

```
public interface IRobozillaSafe
{
    void ClimbStairs();
    void PetTheRobodog();
}
```

2. **Теперь модифицируем класс `Robozilla` как реализующий интерфейс. Поскольку он уже содержит реализацию требуемых методов, все, что нам надо, — добавить `:IRobozillaSafe` в заголовок класса:**

```
public class Robozilla : IRobozillaSafe ...
```

Теперь вы можете держать класс `Robozilla` в секрете от всех, предоставив на всеобщее обозрение только интерфейс `IRobozillaSafe`. Дайте своим клиентам средство для инстанцирования нового объекта `Robozilla`, но возвращайте его как ссылку на интерфейс (вот как это делается при помощи статического метода-фабрики, добавленного в класс `Robozilla`):

```
// Создает объект Robozilla, но возвращает только ссылку на
// интерфейс
public static IRobozillaSafe CreateRobozilla(<параметры>)
{
    return (IRobozillaSafe)new Robozilla(<параметры>);
}
```

После этого клиенты могут использовать `Robozilla` примерно так:

```
IRobozillaSafe myZilla = Robozilla.CreateRobozilla(...);
myZilla.ClimbStairs();
myZilla.PetTheRobodog();
```

Это так просто. Посредством интерфейса они могут вызывать те методы `Robozilla`, которые указаны в интерфейсе, и не более того. Но эксперт может разрушить вашу идиллию простым приведением типа:

```
Robozilla myKillaZilla = (Robozilla)myZilla;
```

Однако поступать так нехорошо. Интерфейс имеет ясное предназначение, и те, кто поступают таким образом, просто нарываются на неприятности — в основном в виде трудно обнаруживаемых ошибок в своих программах. В реальной жизни программисты иногда используют такое присваивание при работе со сложным классом `DataSet` в ADO .NET для взаимодействия с базами данных. `DataSet` может возвращать множество таблиц базы данных с записями, таких как таблица `Customers` (заказчики) и таблица `Orders` (заказы). (Современные реляционные базы данных наподобие Oracle и SQL Server содержат *таблицы*, связанные между собой *отношениями*. Каждая таблица содержит множество *записей*, где каждая запись может быть, например, именем или номером заказчика.)

К сожалению, если у вас есть клиент ссылки `DataSet` (даже посредством свойства только для чтения), он в состоянии легко модифицировать то, что модифицировать ему нельзя. Один из способов предотвращения этого заключается в возврате объекта `DataView`, который предоставляет доступ только для чтения. В качестве альтернативы можно создать интерфейс C# для представления подмножества безопасных операций, доступных посредством `DataSet`. Затем можно создать подкласс `DataSet`, скажем `MyDataSet`, который реализует интерфейс. Наконец, клиенту предоставляется способ получения ссылки на интерфейс на имеющийся объект `MyDataSet`, работа с которым относительно безопасна, поскольку выполняется через интерфейс.

СОВЕТ

Часто лучшим решением оказывается отказ от возврата ссылки на коллекцию, поскольку это позволяет кому угодно изменить коллекцию вне создавшего ее класса. Помните, что полученная ссылка может указывать на исходную коллекцию в классе. Именно поэтому `List<T>`, например, предоставляет метод `AsReadOnly()`, который возвращает коллекцию, которую нельзя изменять:

```
private List<string>
        _readWriteNames = ... // Изменяемый член-данные
...
ReadonlyCollection<string> readonlyNames =
    _readWriteNames.AsReadOnly();
return readonlyNames;          // Это безопаснее, чем вернуть
                               // _readWriteNames
```

Хотя интерфейс здесь не используется, назначение данного решения то же самое.

Наследование интерфейсов

Интерфейс C# может "наследовать" методы другого интерфейса. Кавычки я поставил потому, что это не истинное наследование — не важно, на что оно при этом похоже. В приведенном далее коде в заголовке указывается *базовый интерфейс*, очень похожий на базовый класс:

```
interface IRobozillaSafe : IPet // Базовый интерфейс
{
    // Непоказанные здесь методы ...
}
```

С помощью "наследования" интерфейсом `IRobozillaSafe` интерфейса `IPet` вы можете заставить данное подмножество `Robozilla` реализовать собственную "одомашненность" без навязывания неподходящих свойств:

```
class PetRobo : Robozilla,
                IRobozillaSafe // (а также IPet путем наследования
{
    // Реализация операций Robozilla
    // Реализация операций IRobozillaSafe, затем ...
    // реализация операций IPet (требуется унаследованным
    // интерфейсом IPet)
}
...
// Передаем только безопасную ссылку, а не ссылку на сам PetRobo
IPet myPetRobo = (IPet)new PetRobo();
// ... вызываем для объекта методы IPet.
```

Интерфейс `IRobozillaSafe` наследует интерфейс `IPet`. Таким образом, чтобы реализация `IRobozillaSafe` была полной, классы, реализующие этот интерфейс, должны реализовывать и интерфейс `IPet`. Такое наследование — совсем не то же, что и наследование классов. Например, показанный выше класс `PetRobo` может иметь конструктор, но аналога конструктора базового класса для `IRobozillaSafe` или `IPet` не существует. Интерфейсы не имеют конструкторов. Что еще более важно, полиморфизм с интерфейсами не работает. В то время как вы можете вызвать метод подкласса посредством ссылки на базовый класс (полиморфизм классов), подобная операция с интерфейсами не работает: вы не можете вызвать метод производного интерфейса (`IRobozillaSafe`) посредством ссылки на базовый интерфейс (`IPet`).

Хотя наследование интерфейсов не полиморфно в том смысле, в котором полиморфно наследование классов, вы можете передать объект интерфейса производного типа (`IRobozillaSafe`) в качестве параметра с типом базового интерфейса (`IPet`). Это также означает возможность размещения объектов `IRobozillaSafe` в коллекции объектов `IPet`.

Использование интерфейсов для внесения изменений в объектно-ориентированные программы

Интерфейсы представляют собой ключ к объектно-ориентированному программированию, который обеспечивает гибкость при изменениях в программе. Использование интерфейсов позволяет вам просто рассмеяться в лицо новым требованиям к программе.

ЗАПОМНИ!

Вы не боитесь услышать "измените константу"? Когда передаете новую программу пользователям, они вскоре начинают требовать внесения тех или иных изменений. "Пожалуйста, добавьте новую возможность". "Пожалуйста, исправьте эту ошибку". "В программе конкурента RoboWarrior имеет возможность X, так почему ее не имеет Robozilla из вашей программы?" Многие программы долговечны — тысячи программ, в особенности написанных на Fortran или Cobol, служат по 20–30 лет, а то и больше. При этом они подвергались множеству изменений за эти годы. Это придает планированию и проектированию с учетом будущих изменений наивысший приоритет.

Вот небольшой пример. Предположим, что в иерархии классов `Robot` объекты подклассов могут перемещаться тем или иным способом. `Robocat` крадется. `Robozilla` топает. `Robosnake` ползет. Один из способов реализации различных режимов перемещения заключается в использовании наследования: имеется базовый класс `Robot` и абстрактный метод `Move()`. Затем каждый подкласс перекрывает метод `Move()`, по-разному его реализуя:

```
abstract public class Robot
{
    abstract public void Move(int direction, int speed);
    // ...
}

public class Robosnake : Robot
{
    public override void Move(int direction, int speed)
    {
        // Реализация метода Move() - ползание.
        // ... реальный код, который вычисляет углы и изменяет
        // местоположение робозмеи в системе координат ...
    }
}
```

Но предположим, что вам часто поступают запросы на добавление новых типов перемещения к существующим подклассам `Robot`. "Сделайте `Robosnake` скользящей, а не ползущей!" — и вы вынуждены открывать класс `Robosnake` и модифицировать его метод `Move()`.


ЗАПОМНИ!

Поскольку метод `Move()` вполне корректно работает для ползания, большинство программистов предпочтет не вмешиваться в него. Реализация ползанья достаточно сложна, и ее изменение может вызвать новые ошибки. Не надо чинить неполоманное!

Гибкие зависимости через интерфейсы

Существует ли способ реализации `Move()`, который облегчал бы вашу участь при очередном пожелании очередного клиента? Конечно, это интерфейсы! Посмотрите на приведенный далее код, который использует отношение СОДЕРЖИТ между классами:

```
public class Robot
{
    // Этот объект используется для реализации движения
    protected Motor _motor = new Motor(); // Ссылка на Motor
    // ...
}
internal class Motor { ... }
```

Идея заключается в том, что имеется содержащийся в классе конкретный (т.е. не являющийся абстрактным) объект типа `Motor`. Отношение СОДЕРЖИТ определяет зависимость между классами `Robot` и `Motor`: `Robot` зависит от конкретного класса `Motor`. Класс с конкретными зависимостями является тесно связанным: когда вам надо заменить `Motor` чем-то иным, скорее всего, потребуется полная замена кода, зависящего от `Motor`. Вместо этого можно изолировать ваш код, используя зависимость только от открытого интерфейса. Таким образом, зависимость от другого объекта будет существенно ослаблена.

Абстрактный или конкретный? Когда следует использовать абстрактный класс, а когда — интерфейс

В главе 17, "Полиморфизм", я знакомил вас с птицами. Там я говорил, что каждая птица представляет собой некоторый подтип `Bird`. Другими словами, утка представляет собой экземпляр подкласса `Duck`. Вы никогда не увидите экземпляр класса `Bird`: этот класс — чистая *абстракция*. Вы всегда имеете дело с *конкретными*, физическими утками, воробьями или пингвинами. Абстракции представляют собой концепции. В качестве живых птиц утки представляют собой реальные, конкретные объекты. А конкретные объекты представляют

собой экземпляры конкретных классов. (*Конкретный класс* — это класс, который может быть инстанцирован. В нем не используется ключевое слово `abstract`, и он реализует все методы.)

ЗАПОМНИ!

В C# абстракции можно представить двумя способами: с помощью абстрактных классов или с помощью интерфейсов. Эти способы отличаются один от другого, и эти отличия могут влиять на ваш выбор.

- **Используйте абстрактный класс**, когда имеется определенная реализация, которая может с пользой применяться подклассами — абстрактный базовый класс может иметь определенный реальный код, наследуемый подклассами. Например, класс `Robot` может обслуживать часть функциональности роботов, не связанную с перемещением.

  Абстрактный класс не обязан быть полностью абстрактным. Жесткое требование — наличие как минимум одного абстрактного, не реализованного метода или свойства, в то время как прочие могут быть реализованы (иметь тела). Использование абстрактного класса для предоставления определенной реализации своим подклассам путем наследования позволяет избежать дублирования кода, что всегда неплохо.

- **Используйте интерфейс**, когда у вас нет никакой реализации для общего использования или когда ваш реализующий класс уже имеет базовый класс.

  Интерфейсы C# полностью абстрактны. Интерфейс C# не предоставляет никакой реализации ни одного из своих методов. Он может также внести дополнительную гибкость, невозможную иначе. Использование абстрактного класса может оказаться невозможным в связи с тем, что вы захотите добавить новые возможности к классу, имеющему базовый класс (исходный текст которого вы не можете изменять). Например, класс `Robot` может уже иметь базовый класс из библиотеки, написанной не вами, который вы, само собой, не в состоянии изменить. Интерфейсы в особенности хорошо подходят для представления полностью абстрактных возможностей, таких как возможности перемещения или вывода, которые вы намерены добавить к нескольким классам, кроме этого не имеющим ничего общего один с другим.

Реализация отношения СОДЕРЖИТ с помощью интерфейсов

Ранее упоминалось, что вы можете использовать интерфейсы в качестве более общего ссылочного типа. Содержащий класс может ссылаться на содержащийся не через ссылку на конкретный класс, а через ссылку на абстракцию — либо на абстрактный класс, либо на интерфейс C#:

```
AbstractDependentClass dependency1 = ...;
ISomeInterface dependency2 = ...;
```

Предположим, что у вас есть интерфейс `IPropulsion`:

```
interface IPropulsion
{
    void Movement(int direction, int speed);
}
```

Класс `Robot` может содержать данные-член типа `IPropulsion` вместо конкретного типа `Motor`:

```
public class Robot
{
    private IPropulsion _propel; // Обратите внимание на
                  // использование здесь типа интерфейса
    // Каким-то образом во время выполнения сюда передается
    // конкретный объект силовой установки...

    // Прочие члены... затем:
    public void Move(int speed, int direction)
    {
        // Использование конкретной силовой установки из
        // данных-члена _propel
        _propel.Movement(speed, direction); // Делегирование
                                            // методу _propel
    }
}
```

Метод `Move()` класса `Robot` делегирует реальную работу объекту, на который ссылается как на интерфейс. От вас требуется предоставить способ присваивания конкретного объекта `Motor`, `Engine` или иной реализации `IPropulsion` данным-члену `_propel`. Программисты зачастую передают такие объекты в качестве параметров конструктора

```
Robot r = new Robosnake(someConcreteMotor); // Тип IPropulsion
```

либо присваивают его при помощи свойства

```
r.PropulsionDevice = someConcreteMotor; // Используется свойство
```

Еще один подход состоит в использовании *метода-фабрики*, о чем говорилось ранее в этой главе, в разделах “Тип, возвращаемый методом” и “Что скрыто за интерфейсом”:

```
IPropulsion _propel = CreatePropulsion(); // Метод-фабрика
```

Глава 19

Делегирование событий

В ЭТОЙ ГЛАВЕ...

- Использование делегатов для решения задачи обратного вызова
- Использование делегатов для настройки метода
- Использование анонимных методов
- Создание событий C#

Эта глава завела нас в тот закуток C#, который присутствовал в этом языке программирования изначально. Возможность *обратного вызова* (callback), метода, используемого для обработки событий, очень важна для приложений C# всех видов. Фактически в настоящее время обратный вызов используется во всех видах приложений. Даже веб-приложения, чтобы работать должным образом, должны иметь некоторый механизм обратного вызова.

Альтернативой является приостановка приложения в ожидании чего-то. Это означает, что приложение не будет реагировать ни на что, кроме ожидаемого ввода. Именно так работают консольные приложения, рассмотренные к этому моменту. Вызов `Console.Read()`, по существу, останавливает приложение, пока пользователь не выполнит некоторый ввод. Консольное приложение может работать таким образом, но если вы хотите, чтобы пользователь мог

нажимать кнопки на форме, у вас должно быть что-то получше, а именно — механизм обратного вызова. В C# вы реализуете обратный вызов с помощью *делегата*, который является описанием того, что требуется методу обратного вызова для обработки события. Делегат действует как тип ссылки на метод. В дополнение к методам обратного вызова эта глава также поможет вам понять, как создавать и использовать делегатов.

Звонок домой: проблема обратного вызова

Если вы видели фильм Стивена Спилберга *Инопланетянин*, то помните, как маленький уродливый, но такой симпатичный пришелец, оказавшийся на Земле, пытается собрать из частей старых игрушек аппарат, чтобы позвонить домой и попросить выслать за ним корабль.

Немного необычное вступление, правда? Но коду C# тоже иногда требуется позвонить домой... Например, вас никогда не интересовало, как работает индикатор хода выполнения задания (progress bar) в Windows? Такая горизонтальная полоска, быстро бегущая (но гораздо чаще — медленно ползущая) и показывающая, какая часть длительной работы (типа копирования файлов) выполнена. Работа индикатора основана на том, что длинные операции время от времени "звонят домой", что на программистском языке называется *обратным вызовом* (callback). Обычно длинные операции оценивают, какое время им потребуется для завершения всей работы, а затем постоянно отслеживают, какая часть работы выполнена. Периодически при помощи *метода обратного вызова* (callback method) они посылают сигнал "на базу" — классу, который инициировал выполнение длинной операции. Этот класс и обновляет свой индикатор. Вся хитрость в том, что вы должны предоставить длинной операции этот метод обратного вызова.

Метод обратного вызова может принадлежать тому же классу, что и длинная операция, — ну, как если бы вы звонили жене из спальни на кухню. Но чаще встречается ситуация, когда индикатором занимается другой класс, что можно приравнять к междугородному звонку любимой тетушке. Иногда в начале работы длинная операция предоставляет механизм для звонка, как если бы вы дали своему ребенку мобилку, чтобы иметь возможность созвониться с ним. В этой главе будет рассказываться о том, как ваш код может настроить механизм обратного вызова и, когда это необходимо, выполнить звонок домой.


ЗАПОМНИ!

Механизм обратного вызова регулярно используется в программировании для Windows, обычно в коде вашей программы, который должен уведомлять модуль на более высоком уровне о завершении работы над заданием, запрашивать необходимые данные или позволять

модулю выполнять некоторые полезные действия, такие как запись в журнал или обновление индикатора. Местом, где вы чаще всего используете обратные вызовы, является пользовательский интерфейс. Когда пользователь что-то делает с пользовательским интерфейсом, например нажимает кнопку, он генерирует событие. Метод обратного вызова это событие обрабатывает.

Определение делегата

Для выполнения обратных вызовов C# предоставляет *делегаты* (delegates), а также ряд иных вещей. Делегаты представляют собой способ C# (вообще-то, способ .NET, поскольку их может использовать любой язык .NET) передачи методов, как если бы они были данными. Вы говорите "когда потребуется, выполни этот метод" (и передаете метод для выполнения). Эта глава поможет вам разобраться в упомянутой концепции, понять ее полезность и начать использовать в своих программах.

Вы можете быть опытным программистом, который тут же обнаружит схожесть делегатов с указателями на функции C/C++, — только делегаты существенно лучше. Но в этом разделе я предполагаю, что среди читателей таких программистов нет.

Рассматривайте делегат как транспорт для передачи метода обратного вызова некоторому методу — "рабочей лошадке", — которому требуется осуществлять такие вызовы или требуется помощь при выполнении определенных действий, например, для обработки каждого элемента коллекции (поскольку коллекции ничего не известно о пользовательских действиях, требуется способ передачи этого действия коллекции для выполнения). На рис. 19.1 показано, как части такой схемы взаимодействуют между собой.

Делегат представляет собой тип данных, подобный классу. Как и для класса, для его использования вы создаете экземпляр типа делегата. На рис. 19.1 показана последовательность событий жизненного цикла делегата.

1. **Определение типа данных делегата (так же, как вы определяете класс).**

   Иногда в C# имеются готовые делегаты, которые вы можете использовать. Однако гораздо чаще требуется определять собственные пользовательские делегаты.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Технически делегат представляет собой класс, производный от класса `System.MulticastDelegate`, который в состоянии хранить один или несколько "указателей" на методы и вызывать их для вас. Можете расслабиться — компилятор сам создаст всю необходимую часть класса для вас.

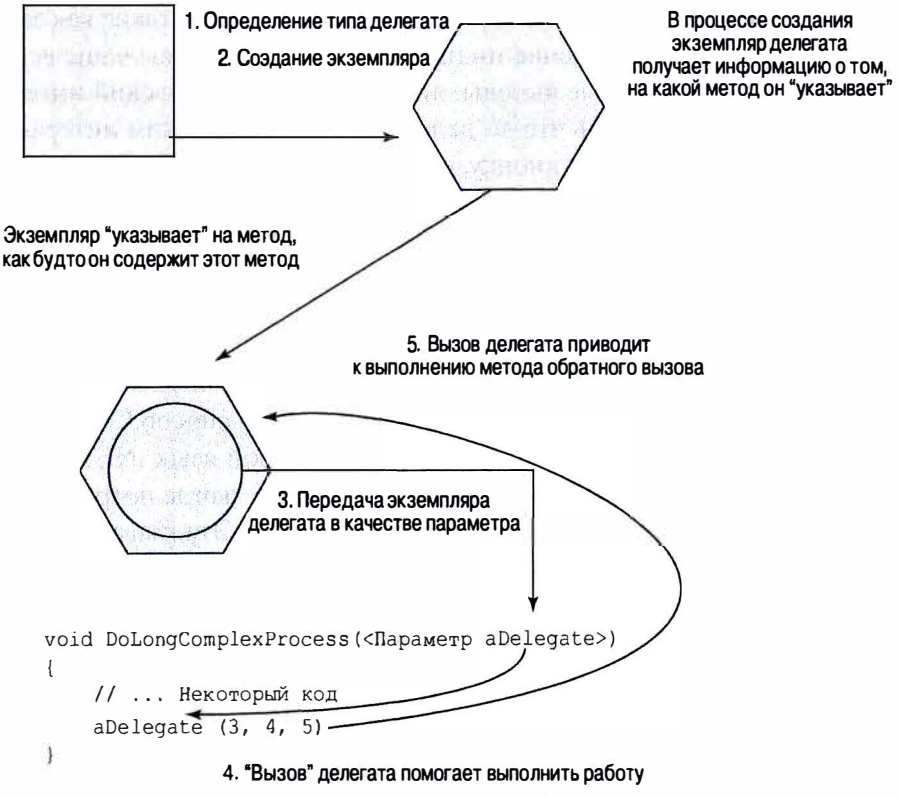


Рис. 19.1. Схема передачи делегата

2. **Создание экземпляра типа делегата аналогично созданию экземпляра класса.**


ЗАПОМНИ!

В процессе создания вы передаете новому экземпляру делегата имя метода, который хотите использовать в качестве метода обратного вызова или метода действия.

3. **Передача экземпляра делегата некоторому рабочему методу, который имеет параметр с типом делегата.**

   Это тот проход, через который вы передаете экземпляр делегата рабочему методу. Это как контрабанда своего попкорна в кинотеатр — с тем отличием, что в этом кинотеатре этот попкорн не просто ожидается, а приветствуется.

4. **Когда рабочий метод готов — например, когда наступает время обновить индикатор хода выполнения, — он вызывает делегат, передавая ему необходимые аргументы.**

5. **Вызов делегата, в свою очередь, выполняет метод обратного вызова, на который "указывает" этот делегат.**

   Используя делегат, рабочий метод "звонит домой".

Этот фундаментальный механизм не только решает задачу обратного вызова, но и имеет иные применения. Типы делегатов могут быть обобщенными, позволяющими использовать один и тот же делегат для различных типов данных, так же, как вы можете инстанцировать коллекцию `List<T>` для элементов типа `string` или `int`.

Пример передачи кода

В этом разделе мы рассмотрим пример решения поставленной в начале этой главы задачи.

Делегирование задания

В этом разделе я приведу два примера применения обратного вызова — когда экземпляр делегата звонит домой, объекту, создавшему его. Но сначала давайте рассмотрим несколько распространенных ситуаций, в которых можно использовать делегат обратного вызова.

- **Уведомление базы делегата о событии**: завершена некоторая длинная операция или сделана определенная часть работы, или произошла ошибка. "Привет, я уже допил свое пиво в баре. Подъедь и забери меня отсюда, а то идти самостоятельно я уже не могу..."
- **Звонок на базу за дополнительными данными, необходимыми для завершения задачи**. "Привет, мам! Я уже в магазине! Я беру два кило конфет, но забыл, сколько я должен был взять картошки — 100 или 200 граммов?"


ЗАПОМНИ!

- **Делегаты позволяют настроить метод**. Настраиваемый метод представляет собой схему, в то время как вызывающий метод предоставляет делегат для выполнения работы. "Я же дала тебе список того, что надо купить, — вытащи его из кармана и покупай все в точности так, как я написала!" Метод делегата выполняет задачу, которую должен выполнить настраиваемый метод (но который не в состоянии сделать это самостоятельно). Настраиваемый метод отвечает за вызов делегата в нужный момент.

Очень простой первый пример

Программа `SimpleDelegateExample` демонстрирует применение очень простого делегата. Код, связанный с работой делегата, выделен полужирным шрифтом.

```
// SimpleDelegateExample - демонстрация очень простого
// обратного вызова делегата
using System;
namespace SimpleDelegateExample
{
    class Program
    {
        delegate int MyDelType(string name); // Внутри класса
                                             // или пространства имен
        static void Main(string[] args)
        {
          // Создание экземпляра делегата, указывающего на метод
          // CallBackMethod, приведенный ниже. Обратите внимание
          // на то, что метод обратного вызова статический,
          // поэтому мы предваряем его имя именем класса —
          // Program.
          MyDelType del = new MyDelType(Program.CallBackMethod);

          // Вызов метода, который будет вызывать делегата
          UseTheDel(del, "hello");

          // Ожидаем подтверждения пользователя
          Console.WriteLine("Нажмите <Enter> для " +
                            "завершения программы...");
          Console.Read();
        }

        // UseTheDel - рабочий метод, который получает в
        // качестве аргумента делегат MyDelType и вызывает его.
        // arg представляет собой строку, которую следует
        // передать делегату при вызове
        private static void UseTheDel(MyDelType del, string arg)
        {
            if (del == null) return; // Нулевой делегат
                                     // не вызывается!
            // Вызов делегата
            // Здесь выводим число, представляющее собой длину arg.
            Console.WriteLine("UseTheDel пишет {0}", del(arg));
        }

        // CallBackMethod - метод, который отвечает сигнатуре
        // делегата MyDelType (получает string, возвращает int).
        // Делегат будет вызывать этот метод.
        public static int CallBackMethod(string stringPassed)
        {
            // Здесь выводится переданная строка.
            Console.WriteLine("CallBackMethod пишет: {0}",
                              stringPassed);
            // Возвращает int.
            return stringPassed.Length; // Делегат требует
                                        // возврата int.
        }
    }
}
```

Вначале вы видите определение делегата. `MyDelType` определяет *сигнатуру* — вы можете передать делегату любой метод, который принимает аргумент типа `string` и возвращает значение типа `int` (метод `CallBackMethod()`, определенный в нижней части листинга, отвечает этой сигнатуре). Метод `Main()` создает экземпляр делегата `del`, а затем передает этот экземпляр рабочему методу `UseTheDel()` вместе с некоторыми строковыми данными (`"hello"`), требующимися делегату. Вот как выглядит последовательность событий при выполнении программы.

1. Метод `UseTheDel()` получает два аргумента, делегат `MyDelType` и строку `string` с именем `arg`. Когда `Main()` вызывает `UseTheDel`, он передает экземпляр делегата для использования в этом методе. При создании экземпляра делегата `del` в методе `Main()` ему передается *имя* метода `CallBackMethod()` как метода, который будет вызван. Поскольку метод `CallBackMethod()` статический, его имя должно быть предварено именем класса, т.е. именем `Program`.
2. В методе `UseTheDel()` проверяется, что делегат не является значением `null`, а затем вызывается метод `WriteLine()`. В этом вызове выполняется запуск делегата `del(arg)`; `arg` — это аргумент, который передается делегату. Этот вызов приводит к вызову метода `CallBackMethod()`.
3. Внутри `CallBackMethod()` метод выводит собственное сообщение, включающее переданную при запуске делегата строку. Затем метод `CallBackMethod()` возвращает длину переданной строки, которая выводится в последней части `UseTheDel()`.

Вывод программы имеет следующий вид:

```
CallBackMethod пишет: hello
UseTheDel пишет 5
Нажмите <Enter> для завершения программы...
```

Метод `UseTheDel()` звонит домой, а `CallBackMethod()` отвечает на звонок.

Более реальный пример

Чтобы быть более реалистичным, я решил написать небольшое приложение, выводящее индикатор хода выполнения задания, и обновлять его всякий раз, когда длинный метод вызывает делегат. Коды примеров из этой главы можно найти в папке `\CSAIO4D\BK02\CH09` загружаемых исходных текстов, о которых говорилось во введении.

Обзор большего примера

Пример программы `SimpleProgress` на веб-сайте книги демонстрирует управляющий элемент `ProgressBar`, о котором я говорил в начале этой главы.

(Кстати, это единственный пример *графической* программы для Windows в данной книге.)

Данная программа выводит небольшое диалоговое окно с двумя кнопками и индикатором хода выполнения задания. После того как вы соберете программу в Visual Studio, запустите ее и щелкнете на верхней кнопке (Click to Start), индикатор начнет движение слева направо, по одной десятой своей длины на каждом шаге. По завершении работы вы можете либо заново запустить его, либо щелкнуть на кнопке Close для завершения программы.

Создание приложения

Чтобы самостоятельно создать приложение и получить опыт в графическом программировании для Windows, выполните перечисленные ниже действия. Сначала надо создать проект и разместить необходимые управляющие элементы в вашем "окне".

1. **Выберите File⇨New Project (Файл⇨Новый проект) и в левой части выберите тип проекта (в этот раз вместо консольного приложения следует выбрать Windows Classic Desktop (классическое настольное приложение Windows)).**

   Вы увидите список потенциальных приложений, показанный на рис. 19.2.

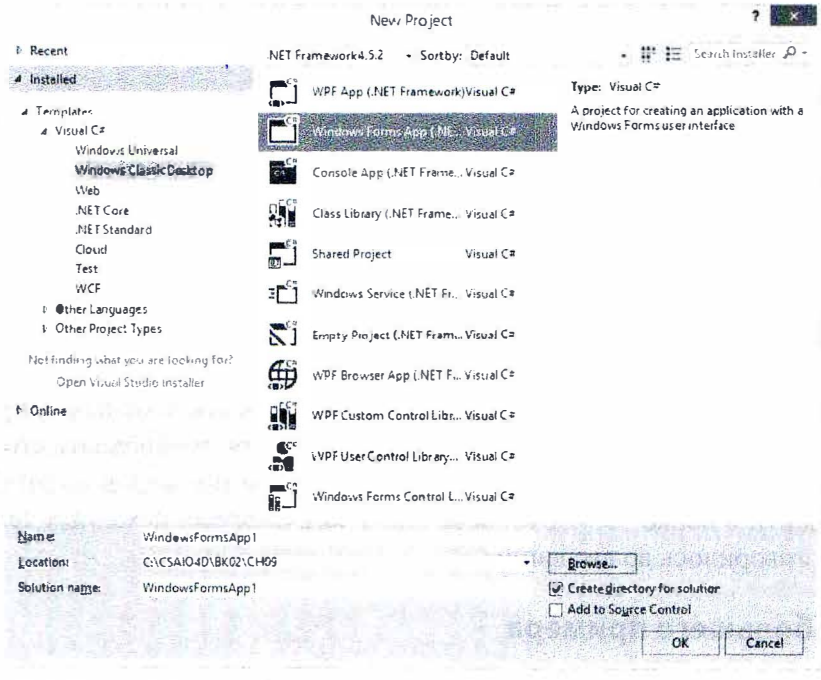


Рис. 19.2. Папка Windows Classic Desktop содержит старые типы проектов

2. **Выберите из списка Windows Forms App (приложение Windows Forms).**
3. **Введите `SimpleProgress` в поле имени проекта Name и щелкните на кнопке OK.**

   Первое, что вы увидите после этого на экране, — *форма*, окно, в котором вы будете размещать *управляющие элементы*.
4. **Выберите пункт меню View⇨Toolbox (Вид⇨Панель элементов) и откройте группу основных управляющих элементов Common Controls (Основные элементы).**

   Вы увидите список управляющих элементов, показанный на рис. 19.3.

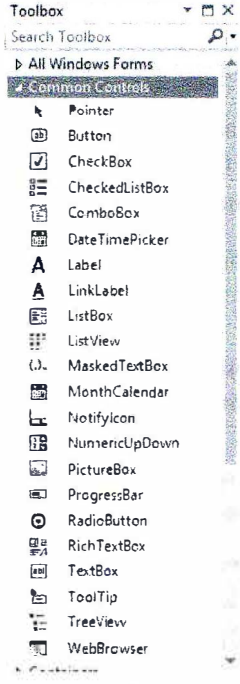


Рис. 19.3. Группа Common Controls содержит наиболее часто используемые управляющие элементы

5. **Перетащите управляющий элемент ProgressBar (индикатор) на форму; затем перетащите туда же два управляющих элемента Button (кнопка).**
6. **Разместите управляющие элементы на форме таким образом, как показано на рис. 19.4.**

   Обратите внимание на направляющие линии, облегчающие процесс размещения.

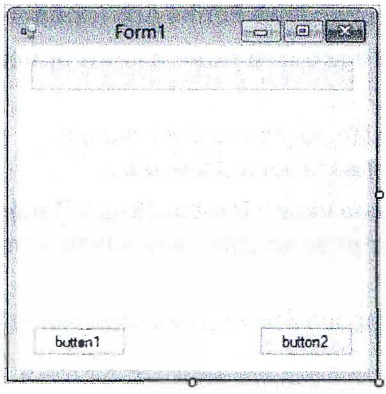


Рис. 19.4. Форма для демонстрации использования индикатора хода выполнения

Далее надо указать свойства размещенных управляющих элементов. Выберите View⇨Properties (Вид⇨Свойства), выберите управляющий элемент на форме и установите следующие свойства.

1. Для индикатора — который в коде имеет имя `progressBar1` — следует установить свойство `Minimum` равным 0, `Maximum` равным 100, `Step` — 10, а `Value` — 0.
2. Для кнопки `button1` измените свойство `Text` на `"Click to Start"` и измените размер кнопки так, чтобы она имела правильный вид и весь текст помещался на ней полностью.
3. Для кнопки `button2` измените свойство `Text` на `"Close"` и измените размер кнопки так, чтобы она имела правильный вид.

СОВЕТ

В этом простом примере весь код находится в классе *формы* (форма представляет собой ваше окно; ее класс — здесь он имеет имя `Form1` — отвечает за весь графический вывод). В общем случае лучше поместить весь "бизнес"-код — т.е. код, который выполняет все вычисления, обращение к данным и прочую важную работу — в другой класс. Класс формы лучше зарезервировать для кода, который связан исключительно с выводом формы и отвечает на ее управляющие элементы. Сейчас я нарушаю это правило, но делегат работает независимо от того, где находится метод обратного вызова.

Теперь добавим к каждой кнопке *метод обработчика* (handler method).

1. **Дважды щелкните на кнопке Close.**

   Это приведет к генерации метода в коде формы. Он должен выглядеть следующим образом (полужирным шрифтом выделен исходный текст, добавленный вами):

```
private void button2_Click(object sender, EventArgs e)
{
    Close();
}
```


СОВЕТ

Для переключения между кодом и изображением формы выберите просмотр кода View⇨Code или просмотр проекта View⇨Designer.

2. **Дважды щелкните на кнопке Click to Start для генерации ее обработчика**:

```
private void button1_Click(object sender, EventArgs e)
{
    UpdateProgressCallback callback =
        UpdateProgressCallback(this.DoUpdate);
    // Некоторая работа, от которой требуется периодическое
    // информирование о ходе выполнения. Выполняется передача
    // экземпляра делегата, который знает, как обновляется
    // индикатор
    DoSomethingLengthy(callback);
    // Сбрасываем индикатор для того, чтобы его можно было
    // использовать повторно.
    progressBar1.Value = 0;
}
```

Вы видите красные подчеркивающие линии под `UpdateProgressCallback` и `DoSomethingLengthy`, которые указывают на ошибки. Пока что игнорируйте эти ошибки — на последующих шагах они будут исправлены.

3. **Добавьте к классу формы метод обратного вызова**:

```
private void DoUpdate()
{
  progressBar1.PerformStep(); // Требует от индикатора
                              // самостоятельно обновиться.
}
```

В следующем разделе мы рассмотрим остальной код программы, а позже — другие варианты передачи делегата.

Знакомимся с кодом

Остальная часть кода из класса `Form1` представляет собой жизненный цикл делегата. Код класса приведен ниже; полужирным шрифтом выделен добавленный код (кроме уже рассмотренного выше).

```
using System;
using System.Windows.Forms;
namespace SimpleProgress
{
    public partial class Form1 : Form
    {
        // Объявление делегата. Он имеет тип void.
        delegate void UpdateProgressCallback();
```

```
        public Form1()
        {
            InitializeComponent();
        }

        // DoSomethingLengthy - рабочий метод, который в
        // качестве параметра получает делегат.
        private void
        DoSomethingLengthy(UpdateProgressCallback updateProgress)
        {
            int duration = 2000;
            int updateInterval = duration / 10;

            for (int i = 0; i < duration; i++)
            {
                Console.WriteLine("Некие действия");

                // Обновление по достижении каждой десятой общей
                // продолжительности.
                if ((i % updateInterval) == 0 &&
                    updateProgress != null)
                {
                    updateProgress(); // Вызов делегата.
                }
            }
        }

        // DoUpdate - метод обратного вызова.
        private void DoUpdate()
        {
            progressBar1.PerformStep();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            // Инстанцирование делегата, указание метода, который
            // должен вызываться.
            UpdateProgressCallback callback =
                new UpdateProgressCallback(this.DoUpdate);
            // Некоторая работа, от которой требуется
            // периодическое информирование о ходе выполнения.
            // Выполняется передача экземпляра делегата, который
            // знает, как обновляется индикатор
            DoSomethingLengthy(callback);
            // Сбрасываем индикатор для того, чтобы его можно было
            // использовать повторно.
            progressBar1.Value = 0;
        }
        private void button2_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }
}
```

ЗАПОМНИ!

Объявление класса немного отвлечет нас от основного изложения:

```
public partial class Form1 : Form
```

Ключевое слово `partial` указывает, что это только часть полного класса. Остальная часть класса может быть найдена в файле `Form1.Designer.cs`, который имеется в списке в обозревателе решений. Позже в данной главе я еще раз обращусь к этому файлу, чтобы проиллюстрировать "события". Частичные классы были введены в C# 2.0. Они позволяют разбить класс между двумя и более файлами. Компилятор генерирует файл `Form1.Designer.cs`, так что нельзя непосредственно изменять код, содержащийся в нем. Его можно модифицировать только опосредованно. Сказанное не относится к файлу `Form1.cs`, который представляет собой *вашу* часть кода.

Жизненный цикл делегата

Теперь рассмотрим код, который представляет собой различные части жизненного цикла делегата.

1. **Делегат `UpdateProgressCallback` определяется в начале класса:**

   ```
   delegate void UpdateProgressCallback();
   ```

   Метод, на который может "указывать" данный делегат, должен не иметь параметров и не должен возвращать значение. После ключевого слова `delegate` указывается *сигнатура* метода, на который может указывать делегат, т.е. его возвращаемый тип и количество, порядок и типы параметров. Делегаты не обязаны быть `void` — вы можете написать делегаты, которые возвращают любой тип и принимают любые аргументы.

   Объявление делегата определяет *тип*, как это делает, например, объявление `class Student {...}`. Вы можете объявить делегат как `public`, `internal`, `protected` или даже при необходимости как `private`.

СОВЕТ

   Неплохо добавить к имени типа делегата `Callback` — само собой, это просто совет, но никак не требование языка программирования.

2. **Создается экземпляр делегата и передается методу `DoSomethingLengthy()` в методе `button1_Click()`:**

   ```
   UpdateProgressCallback callback = // Инстанцирование делегата.
           new UpdateProgressCallback(this.DoUpdate);
   DoSomethingLengthy(callback);      // Передача экземпляра
                                      // делегата методу.
   ```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

   Этот делегат "указывает" на метод класса `this` (`this` в данном случае писать не обязательно). Для указания на метод другого класса необходим экземпляр этого класса (если это метод экземпляра), и метод передается следующим образом:

```
SomeClass sc = new SomeClass();
UpdateProgressCallback callback =
        new UpdateProgressCallback(sc.DoUpdate);
```

Но если это статический (`static`) метод, то передача его осуществляется так:

```
UpdateProgressCallback callback =
        new UpdateProgressCallback(SomeClass.DoUpdate);
```

ЗАПОМНИ!

При создании экземпляра передается только *имя* метода, но не его параметры. Методу `DoSomethingLengthy()` передается экземпляр делегата `callback` (который указывает на метод).

3. **Ваш метод `DoSomethingLengthy()` выполняет некоторую "долгую работу" и периодически приостанавливается, чтобы при помощи метода обратного вызова сообщить форме, что она может обновить индикатор хода выполнения задания.**

Вызов делегата в методе `DoSomethingLengthy()` выглядит так же, как и вызов метода (при необходимости — с передачей параметров):

```
updateProgress(); // Вызов переданного экземпляра делегата
```

Метод `DoSomethingLengthy()` имеет следующий вид:

```
private void
    DoSomethingLengthy(UpdateProgressCallback updateProgress)
{
    int duration = 2000;
    int updateInterval = duration / 10;
    for (int i = 1; i <= duration; i++)
    {
        Console.WriteLine("Некие действия");
        // Периодическое обновление формы.
        if ((i % updateInterval) == 0 && updateProgress != null)
        {
            updateProgress(); // Вызов делегата.
        }
    }
}
```

Наш "длинный процесс" на самом деле ничего такого важного не делает. Он устанавливает переменную `duration` равной 2000 итераций цикла — несколько секунд времени выполнения, этого более чем достаточно для демонстрационной программы. Затем метод вычисляет "интервал обновления" в 200 итераций путем деления общей продолжительности на десять. После этого цикл `for` выполняет эти 2000 итераций. На каждой из них цикл проверяет, не пора ли обновить интерфейс. В большинстве случаев никакое обновление не выполняется. Но когда условие `if` становится истинным, метод вызывает экземпляр `UpdateProgressCallback`, который был передан ему в качестве параметра `updateProgress`. Выражение `i%updateInterval`, которое представляет собой получение остатка от деления, становится равным 0 (т.е. соответствует условию `if`) каждые 200 итераций.

Перед вызовом делегата надо всегда проверять, не равен ли он `null`.

4. **Когда метод `DoSomethingLengthy()` вызывает делегат, тот, в свою очередь, вызывает метод, на который указывает; в данном случае это метод `DoUpdate()` класса `Form1`.**

5. **При вызове с помощью делегата метод `DoUpdate()` выполняет обновление при помощи вызова метода `PerformStep()` класса `ProgressBar`:**

```
private void DoUpdate()
{
    progressBar1.PerformStep();
}
```

Метод `PerformStep()`, в свою очередь, заполняет цветом очередные 10% полосы индикатора — величину, определяемую свойством `Step`, установленным равным 10.

6. **Управление возвращается методу `DoSomethingLengthy()`, который продолжает выполнение цикла. По завершении цикла выполняется выход из метода `DoSomethingLengthy()` и возврат управления методу `button1_Click()`. Этот метод очищает индикатор `ProgressBar`, устанавливая его свойство `Value` равным 0. После этого приложение дожидается очередного щелчка на одной из кнопок (или пиктограмме закрытия приложения в правом верхнем углу окна).**

Вот и все. Используя делегат для реализации обратного вызова, программа поддерживает актуальность состояния индикатора завершенности выполнения задания. Если вам необходимо определить тип делегата с параметрами для реализации обратного вызова, вы можете разработать собственный делегат. Для событий и методов `Find()` и `ForEach()` классов коллекций можно воспользоваться *предопределенными* делегатами.

Анонимные методы

После того как вы осознаете суть использования делегатов, взгляните на первое упрощение работы с делегатами в C# 2.0. Чтобы уменьшить количество канители при работе с делегатом, можно использовать анонимный метод. Анонимные методы просто записываются более традиционным способом и, хотя синтаксис и некоторые детали различаются, результат, по сути, оказывается одинаковым независимо от того, используете ли вы необработанный делегат, анонимный метод или лямбда-выражение.

Анонимный метод одновременно создает экземпляр делегата и метод, на который он "указывает", прямо "на лету". Вот как выглядят "внутренности" метода `DoSomethingLengthy()` при применении анонимного метода (см. текст, выделенный полужирным шрифтом):

```
private void DoSomethingLengthy() // Аргументы в этот раз
{                                 // нам не нужны
    ...
    for (int i = 0; i < duration; i++)
    {
        if ((i % updateInterval) == 0)
        {
            // Создание экземпляра делегата
            UpdateProgressCallback anon = delegate()
            {
                // Метод, на который "указывает" делегат
                progressBar1.PerformStep();
            };
            if(anon != null) anon();       // Вызов делегата
        }
    }
}
```

Код выглядит так же, как и рассматривавшиеся ранее инстанцирования делегатов, за исключением того, что после знака = идет ключевое слово `delegate`, в круглых скобках — необходимые параметры, передаваемые анонимному методу (если таких параметров нет, скобки остаются пустыми), и тело метода. Код, который ранее был в отдельном методе `DoUpdate()` — методе, на который "указывал" делегат, — теперь перемещается в анонимный метод. "Указание" на метод больше не используется, а сам метод не имеет имени. Вам все еще требуются определение типа делегата `UpdateProgressCallback` и вызов экземпляра делегата (в данном примере — `anon`).

Можно не упоминать, что приведенный здесь материал не охватывает всего, что следует знать об анонимных методах, — это только начало. Поищите *anonymous method* в справочной системе, а также познакомьтесь с программой `DelegateExamples` на веб-сайте книги. Еще один совет: ваши анонимные методы должны быть как можно короче.

События C#

Далее будет рассмотрено еще одно применение делегатов — это *события* C#, которые реализуются при помощи делегатов. События представляют собой разновидность обратных вызовов, но обеспечивают более простой по сравнению с обратными вызовами механизм уведомления о наступлении важных событий. Особенно полезны события, когда обратный вызов ожидается несколькими методами. События широко используются в C#, в особенности для связи объектов пользовательского интерфейса с кодом, который заставляет их работать. Примером могут служить кнопки из рассмотренного ранее примера `SimpleProgress`.

Проектный шаблон Observer

В программировании весьма распространены ситуации, когда различные объекты программы "интересуются" событиями, происходящими с другими объектами. Например, форма, на которой расположена кнопка, "хочет" знать, когда пользователь щелкает на этой кнопке. События предоставляют в C# и .NET стандартный механизм для уведомления заинтересованных объектов о важных действиях.


СОВЕТ

Проектный шаблон с использованием событий используется так часто, что получил собственное имя — "Наблюдатель" (Observer). Это один из множества других распространенных *проектных шаблонов*, которые любой может применять в собственных программах. Если вас интересуют другие шаблоны, обратитесь к специальной литературе, например к книге Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* (СПб.: Питер, 2001).

Шаблон Observer состоит из наблюдаемого объекта — объекта с интересующими событиями (иногда его называют субъектом) — и произвольного количества наблюдающих объектов, которые заинтересованы в информации о некотором конкретном событии. Наблюдатели некоторым способом регистрируются у наблюдаемого объекта, и, когда происходит интересующее их событие, наблюдаемый объект уведомляет об этом всех зарегистрированных наблюдателей. Реализовать этот шаблон можно множеством различных способов без привлечения событий (таких, как обратные вызовы и интерфейсы), но способ C# состоит в использовании событий.


СОВЕТ

Вместо "наблюдатели" вы можете встретить альтернативное название — "слушатели" (listeners). Слушатели "слушают" события. И это не единственное альтернативное название, есть и другие.

Что такое событие. Публикация и подписка

Одной из аналогий событий является газета. Вы и многие другие люди подписываетесь на газету, и после этого ее доставляют в ваш почтовый ящик. Редакция газеты представляет собой Издателя (Publisher), а читатели — Подписчиков (Subscribers), так что такая вариация шаблона проектирования Observer часто называется *Издание/Подписка* (Publish/Subscribe). Это аналогия, которую я использую в данной главе, но вы не должны забывать, что шаблон Observer *является* шаблоном Publish/Subscribe с иной терминологией. Наблюдатели — это те же подписчики, а наблюдаемый объект — издатель.

Когда в C# у вас имеется класс, в котором происходят интересующие вас события, вы оповещаете о его возможности уведомлять об этом событии другие классы, предоставляя (как правило, открытый) *объект события* (event object).


ЗАПОМНИ!

Термин *событие* в C# имеет два значения. Говоря о событии, можно подразумевать как некоторое явление или действие, так и объект C# определенного вида. Первое является концепцией события в реальном мире, а второе — методикой C#, использующей ключевое слово `event`.

Как издатель оповещает о своих событиях


ЗАПОМНИ!

Чтобы оповестить о возможности подписки, класс объявляет делегат и соответствующее событие примерно следующим образом:

```
public class PublisherOfInterestingEvents
{
    // Тип делегата, на котором базируется событие.
    // Должен быть объявлен как 'internal', если все
    // подписчики находятся в том же пакете.
    public delegate void
        NewEditionEventHandler(object sender,
                               NewEditionEventArgs e);
    // Событие:
    public event NewEditionEventHandler NewEdition;
    // ... Прочий код.
}
```

Определения делегата и события сообщают всему миру: "Подписчики, добро пожаловать!" Событие `NewEdition` можно рассматривать как переменную типа делегата `NewEditionEventHandler`. (Пока что никакие события не рассылаются — это всего лишь инфраструктура для них.)


СОВЕТ

Можно приветствовать добавление суффикса `EventHandler` к имени типа делегата, на котором базируется событие. Само собой, это не стандарт, а всего лишь рекомендация, делающая код более удобочитаемым. Распространенным примером описанной методики является оповещение классом `Button` о различных событиях, включая событие `Click` (см. пример программы `SimpleProgress` на веб-сайте книги). В C# класс `Button` оповещает об этом событии следующим образом:

```
event _dispCommandBarControlEvents_ClickEventHandler Click;
```

Здесь второе длинное слово представляет собой делегат, определенный в недрах .NET.

ЗАПОМНИ!

Из-за широкой распространенности событий .NET определяет два типа делегатов, связанных с событиями — `EventHandler` и `EventHandler<TEventArgs>`. Вы можете заменить `NewEdition EventHandler` в представленном выше коде на `EventHandler` или на обобщенный тип `EventHandler<TEventArgs>`, и вам не потребуется собственный тип делегата. В оставшейся части этой главы будет использоваться встроенный тип делегата `EventHandler<TEventArgs>`, а не `EventHandler` или пользовательский тип `NewEditionEventHandler`. Читателям в своей практике также рекомендуется использовать эту форму делегата:

```
event EventHandler<NewEditonEventArgs> NewEdition;
```

Как подписаться на событие

Для получения информации об определенном событии подписчики должны подписаться:

```
publisher.EventName +=
    new EventHandler<некий тип EventArgs>(некоторое имя метода);
```

Здесь `publisher` представляет собой экземпляр класса издателя, `EventName` — имя события, а `EventHandler<TEventArgs>` — делегат, на котором основано событие. Например, приведенный код может быть следующим:

```
myPublisher.NewEdition +=
       new EventHandler<NewEditionEventArgs>(MyPubHandler);
```

Поскольку объект события за сценой представляет собой делегат, синтаксис `+=` добавляет метод к списку методов, которые будут вызываться делегатом при его вызове.

СОВЕТ

Таким образом, может быть подписано любое количество объектов (и делегат будет при этом хранить список всех подписанных методов) — вплоть до того, что подписаться может даже объект, на событие которого объявлена подписка. (Ну и, конечно, это показывает, что делегат может "указывать" более чем на один метод.) В примере программы `SimpleProgress` в файле `Form1.Designer.cs` можно увидеть, как класс формы регистрирует сам себя в качестве подписчика на события кнопок `Click`.

Как опубликовать событие

Когда издатель решает, что произошло нечто, стоящее того, чтобы уведомить об этом всех подписчиков, он *генерирует* (рассылает) событие. Это

похоже на то, как обычная газета распространяет специальный воскресный выпуск. Для публикации события издатель в одном из своих методов должен иметь код, подобный приведенному далее (см. также раздел "Рекомендованный способ генерации событий" далее в главе):

```
NewEditionEventArgs e =
    new NewEditionEventArgs(<аргументы для конструктора>);
// Генерация события — 'this' представляет собой объект издателя
NewEdition(this, e);
```

В примере `Button` все это скрыто в классе `Button`:

```
EventArgs e = new EventArgs(); // См. следующий раздел
Click(this, e);                // Генерация события
```

В каждом из этих примеров вы используете необходимые аргументы, которые отличаются от события к событию; некоторые из событий требуют передачи большого количества информации. Тогда вы генерируете событие путем "вызова" его имени (подобно вызову делегата):

```
eventName(<список аргументов>); // Генерация события
                                // (доставка газеты)
NewEdition(this, e);
```


ЗАПОМНИ!

События могут базироваться на разных делегатах с различными сигнатурами, имеющими разные параметры, как было показано ранее в примере `NewEditionEventHandler`, но обычно у событий принято предоставление параметров `sender` и `e`. Встроенные типы делегатов `EventHandler` и `EventHandler<TEventArgs>` определяют их для вас.

Передача ссылки на отправитель события (объект, генерирующий событие) оказывается полезной в случае, когда методу обработки события требуется дополнительная информация. Так, конкретный объект `button1` типа `Button` может передать ссылку на класс `Form`, частью которого является эта кнопка. Обработчик события кнопки `Click` находится в классе `Form`, так что отправителем события является форма: методу в качестве аргумента передается значение `this`.


ЗАПОМНИ!

Событие можно генерировать в любом методе класса-издателя. Когда? Генерируйте его тогда, когда это требуется. Генерацию событий мы рассмотрим после следующего короткого раздела.

Как передать обработчику события дополнительную информацию

Параметр e метода обработки события представляет собой пользовательский подкласс класса System.EventArgs. Вы можете написать собственный класс NewEditionEventArgs для передачи необходимой информации:

```
public class NewEditionEventArgs : EventArgs
{
    public NewEditionEventArgs(DateTime date, majorHeadline)
    {
        PubDate = date;
        Head = majorHeadline;
    }
    public DateTime PubDate { get; private set; }
    public string   Head    { get; private set; }
}
```

Вы должны реализовать члены данного класса как свойства, как показано в исходном тексте выше. Закрытые методы установки свойств используются в конструкторе. Часто ваши события не требуют дополнительных аргументов, и вы можете обратиться к базовому классу EventArgs, как показано в следующем разделе. Если для вашего события не нужен специальный объект, порожденный от класса EventArgs, можете написать такой код:

```
NewEdition(this, EventArgs.Empty); // Генерация события.
```

Рекомендованный способ генерации событий

В разделе "Как опубликовать событие" была показана схема генерации события. Однако вы всегда должны определять специальный метод "генерации события" наподобие следующего:

```
protected virtual void OnNewEdition(NewEditionEventArgs e)
{
    EventHandler<NewEditionEventArgs> temp = NewEdition;
    if(temp != null)
    {
        temp(this, e);
    }
}
```

Наличие такого метода гарантирует, что вы не забудете выполнить два шага.

1. **Сохранить событие во временной переменной.**

   Это делает ваше событие более приспособленным к ситуациям, в которых несколько потоков попытаются одновременно использовать его (потоки разделяют вашу программу на приоритетную задачу и одну или несколько фоновых задач, выполняющихся одновременно).

2. **Выполнить проверку на равенство `null` перед тем, как пытаться генерировать событие.**

   Если объект равен `null`, попытка генерации приведет к ошибке. Кроме того, значение `null` говорит о том, что событием никто не интересуется (у него нет подписчиков), так что генерировать его бессмысленно. *Всегда* выполняйте проверку на равенство события `null`, независимо от наличия проверки в методе `On_`*`Событие`*.

Объявляя метод `protected` и `virtual`, вы позволяете подклассам перекрывать его. Это необязательное условие. Если у вас имеется такой метод, который всегда принимает один и тот же вид (что упрощает его написание), вызывайте его всякий раз, когда требуется сгенерировать событие:

```
void SomeMethod()
{
    // Выполнение задач метода, а затем:
    NewEditionEventArgs e =
        new NewEditionEventArgs(DateTime.Today,"Peace Breaks Out!");
    OnNewEdition(e);
}
```

Как наблюдатели "обрабатывают" событие

Подписывающийся объект указывает имя *метода обработки*, который передается в качестве аргумента конструктора (выделен полужирным шрифтом):

```
button1.Click += new EventHandler<EventArgs>(button1_Click);
```

Это примерно то же, что в нашей аналогии с газетой сказать "Высылайте мою газету на такой-то адрес". Вот обработчик нашего события `NewEdition`:

```
myPublisher.NewEdition +=
    new EventHandler<NewEditionEventArgs>(NewEdHandler);

...

void NewEdHandler(object sender, NewEditionEventArgs e)
{
    // Некоторые действия в ответ на событие.
}
```

Например, класс `BankAccount` может генерировать пользовательское событие `TransactionAlert` при каких-либо действиях с объектом `BankAccount` — при внесении денег на счет, снятии со счета, пересылке между счетами и даже при ошибке. Наблюдатель `Logger` может подписаться на это событие и записывать происходящее в файл или в базу данных.

ЗАПОМНИ!

Когда вы создаете в Visual Studio обработчик кнопки (двойным щелчком на кнопке в вашей форме), Visual Studio генерирует код подписки в файле `Form1.Designer.cs`. Вы не должны редактировать подписку, но можете удалить ее и заменить ее таким же кодом, написанным в вашей части частичного класса формы, после чего конструктор форм ничего не будет знать о подписке.

В методе обработки вашего подписчика вы делаете все то, что предполагается делать при получении события данного вида. Чтобы облегчить написание этого кода, вы можете преобразовать параметр `sender` в тип, которым, как вы это знаете, он является:

```
Button theButton = (Button)sender;
```

Затем при необходимости можно вызывать методы и свойства этого объекта. Поскольку у вас есть ссылка на объект-отправитель, вы можете запрашивать у него информацию и выполнять с ним все необходимые операции. Таким же образом можно получить информацию и из параметра `e`: `Console.WriteLine(e.HatSize);`

Вы не обязаны всегда использовать параметры, но иногда это может оказаться очень удобным.

СОВЕТ

КОГДА ЧТО ИСПОЛЬЗОВАТЬ

События. Используйте события, если у вас могут быть несколько подписчиков или при связи с клиентским программным обеспечением, использующим ваши классы.

Делегаты. Используйте делегаты или анонимные методы, если вам требуется обратный вызов или настройка операции.

Лямбда-выражения. Лямбда-выражения — это, по сути, всего лишь краткий способ указать метод, передаваемый делегату. Лямбда-выражения можно использовать вместо анонимных методов.

Глава 20

Пространства имен и библиотеки

В ЭТОЙ ГЛАВЕ...

- Отдельно компилируемые сборки
- Написание библиотек классов
- Ключевые слова управления доступом
- Работа с пространствами имен

Язык C# предоставляет ряд способов разбиения кода на отдельные рабочие модули. Сюда входят методы и классы, которые позволяют разбивать длинные строки кода на легко поддерживаемые модули. Структура класса дает возможность группировать данные и методы для дальнейшего снижения сложности программ. Программы сложны сами по себе и легко приводят в замешательство не только новичков, но и опытных программистов.

Язык C# предоставляет еще один уровень группирования: схожие классы могут быть объединены в библиотеку классов. Кроме создания собственных библиотек, в своих программах вы можете использовать библиотеки, написанные другими программистами. Такие программы состоят из модулей, именуемых *сборками* (assemblies). Эта глава посвящена библиотекам и сборкам.

Начатый в главе 15, "Класс: каждый сам за себя", рассказ об управлении доступом остался незавершенным. Мы так и не рассмотрели ключевые слова `protected`, `internal` и `protected internal`, а также применение *пространств*

имен, которые предоставляют еще один способ группирования подобных классов и позволяют использовать дублирующиеся имена в разных частях программы. Пространства имен также будут рассматриваться в этой главе.

Разделение одной программы на несколько исходных файлов

Программы в данной книге носят исключительно демонстрационный характер. Каждая из них — длиной не более нескольких десятков строк и содержит не более пары классов. Программы же промышленного уровня со всеми "рюшечками" и "финтифлюшечками" могут состоять из сотен тысяч строк кода с сотнями классов.

Рассмотрим систему продажи авиабилетов. У вас должен быть один интерфейс для заказа билетов по телефону, другой — для тех, кто заказывает билет по Интернету, должна быть часть программы, отвечающая за управление базой данных билетов, дабы не продавать один и тот же билет несколько раз, еще одна часть должна следить за стоимостью билетов с учетом всех налогов и скидок, и так далее и тому подобное... Такая программа будет иметь огромный размер. Размещение всех составляющих программу классов в одном исходном файле `Program.cs` быстро становится непрактичным по следующим причинам.

» **Возникают проблемы при поддержке классов**. Единый исходный файл очень трудно поддается пониманию. Гораздо проще разбить его на отдельные модули, например так.

- `Aircraft.cs`
- `Fare.cs`
- `GateAgent.cs`
- `GateAgentInterface.cs`
- `ResAgent.cs`
- `ResAgentInterface.cs`

Это также облегчает задачу поиска интересующего вас конкретного исходного текста.

» **Работа над большими программами обычно ведется группами программистов**. Два программиста не в состоянии редактировать одновременно один и тот же файл — каждому требуется его собственный исходный файл (или файлы). У вас может быть 20 или 30 программистов, одновременно работающих над одним большим проектом. Один файл ограничит работу каждого из 24

программистов над проектом всего одним часом в сутки, но стоит разбить программу на 24 файла, как становится возможным (хотя и сложным) заставить всех программистов трудиться круглые сутки. Разбейте программу так, чтобы каждый класс содержался в отдельном файле, и ваша группа заработает, как слаженный оркестр.

- **Компиляция больших файлов отнимает слишком много времени**. В результате босс начинает нервничать и выяснять, почему это вы так долго пьете кофе вместо того, чтобы стучать по клавишам? Какой смысл перестраивать всю программу, когда кто-то из программистов изменил пару строк кода? Visual Studio 2017 может перекомпилировать только измененный файл и собрать программу из уже готовых объектных файлов.

По всем этим причинам программисты на C# предпочитают разделять программу на отдельные исходные файлы `.cs`, которые компилируются и собираются вместе в единый выполнимый `.exe`-файл.


ЗАПОМНИ!

Файл проекта содержит инструкции о том, какие файлы входят в проект и как они должны быть скомбинированы. Можно объединить файлы проектов для генерации комбинаций программ, которые зависят от одних и тех же пользовательских классов. Например, вы можете захотеть объединить программу записи с соответствующей программой чтения. Тогда, если изменяется одна из них, вторая перестраивается автоматически. Один проект может описывать программу записи, второй — программу чтения. Набор файлов проектов известен под названием *решение* (solution).


СОВЕТ

Программисты на Visual C# используют для объединения нескольких исходных файлов C# в проекты Visual Studio Solution Explorer среды Visual Studio 2017.

Разделение единой программы на сборки

В Visual Studio, а также в C#, Visual Basic .NET и прочих языках .NET один проект соответствует одному скомпилированному *модулю* — в .NET он носит имя *сборка*. Технически модуль и сборка имеют различные значения, но только для опытных программистов. В нашей книге это термины взаимозаменяемые.

Выполнимый файл или библиотека

Язык C# может создавать два основных типа сборок.

- **Выполнимые файлы** (с расширением `.EXE`). Выполнимые файлы представляют собой программы, содержащие метод `Main()`. При двойном щелчке на таком `.EXE`-файле в Проводнике Windows запускается на выполнение программа, хранящаяся в этом файле. В этой книге — множество примеров выполнимых файлов в виде консольных приложений. Выполнимые сборки часто используют код поддержки из библиотек в других сборках.
- **Библиотеки классов** (`.DLL`). Что касается библиотек классов, то, опять же, их используют все программы в книге. Например, пространство имен `System` — место размещения таких классов, как `String`, `Console`, `Exception`, `Math` и `Object` — существует как набор библиотечных сборок. Каждой программе требуются классы `System`. Библиотеки располагаются в DLL-сборках.


ЗАПОМНИ!

Библиотеки не являются самостоятельными выполнимыми программами. Их нельзя запустить на выполнение непосредственно; вместо этого их код вызывается из выполнимых файлов или других библиотек. Поддержка времени выполнения (Common Language Runtime — CLR), которая запускает программы C#, при необходимости загружает библиотечные модули в память.

Важной концепцией, которую необходимо знать, является то, что вы можете легко создавать собственные библиотеки классов. Раздел "Объединение классов в библиотеки" данной главы показывает, как выполнить эту задачу.

Сборки

Сборки представляют собой скомпилированные версии индивидуальных проектов. Они содержат код проектов в специальном формате вместе с *метаданными* — подробной информацией о классах сборки.

Здесь рассказывается о сборках, поскольку они необходимы для полного понимания процесса построения программ C#, а кроме того, эта информация будет важна при рассмотрении пространств имен и ключевых слов, таких как `protected` и `internal`. О пространствах имен и упомянутых ключевых словах мы поговорим немного позже в этой главе. Сборки также играют важную роль при рассмотрении библиотек классов, о чем будет говориться в разделе "Объединение классов в библиотеки".


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Компилятор C# преобразует код проекта C# в код на специальном промежуточном языке (Common Intermediate Language, обычно для него используется аббревиатура "IL"), который хранится в файле сборки. IL — общий язык .NET, т.е. компиляторы всех языков программирования .NET выполняют преобразование исходных

текстов в IL. Этот промежуточный язык напоминает ассемблер — язык, стоящий на один уровень выше машинных кодов, который используют крутые программисты, когда хотят оказаться "поближе к железу" из-за того, что их не устраивают возможности высокоуровневых языков или производительность создаваемых ими программ.

Одно важное следствие из того факта, что все компиляторы .NET выполняют компиляцию исходных текстов в IL, независимо от языка программирования, состоит в том, что можно совместно использовать сборки, написанные на разных языках программирования. Например, программа на C# может вызывать методы из сборки, написанной на Visual Basic или C++, либо программа на C# может порождать подкласс из класса VB.

Выполнимые файлы

Запустить выполнимый файл можно разными способами.

- В среде Visual Studio выбрать пункт меню Debug⇨Start Debugging (Отладка⇨Начать отладку, <F5>) или Debug⇨Start Without Debugging (Отладка⇨Запустить без отладки, <Ctrl+F5>).
- Дважды щелкнуть на `.EXE`-файле в Проводнике Windows.
- Щелкнуть на пиктограмме файла в Проводнике Windows правой кнопкой мыши и выбрать в контекстном меню команду Run (Выполнить) или Open (Открыть).
- Ввести имя (и путь) файла в окне консоли.
- Если программа использует аргументы командной строки, такие как имена файлов, перетащить эти файлы на выполняемый файл в Проводнике Windows.


ЗАПОМНИ!

Решение в Visual Studio может состоять из нескольких проектов — нескольких `.DLL` и нескольких `.EXE`. Если решение содержит более одного `.EXE`-файла, вы должны указать Visual Studio, какой из проектов является *начальным* (startup project). Именно он будет запускаться из меню отладки Debug. Чтобы указать начальный проект, щелкните на нем правой кнопкой мыши в окне обозревателя решений Solution Explorer и выберите в контекстном меню Set as Startup Project (установить в качестве начального проекта). Имя начального проекта будет выделено в Solution Explorer полужирным шрифтом.

Думайте о решении, содержащем две `.EXE`-сборки, как о двух отдельных программах, которые используют одни и те же библиотечные сборки. Например, у вас в решении может быть консольное приложение, приложение

Windows Forms и несколько библиотек. Если вы укажете консольное приложение в качестве начального проекта и скомпилируете код, то получите консольное приложение. Если начальным объявить приложение Windows Forms, то нетрудно догадаться, что вы получите в результате компиляции.

Библиотеки классов

Библиотека классов состоит из одного или нескольких классов (обычно это классы, тем или иным образом сотрудничающие между собой). Зачастую классы в библиотеке находятся в собственном *пространстве имен* (что такое "пространство имен", будет рассказано в разделе "Размещение классов в пространствах имен" немного позже в данной главе). Можно построить библиотеку математических подпрограмм, библиотеку подпрограмм для работы со строками, библиотеку подпрограмм ввода-вывода и т.п.

Случается, что решение целиком представляет собой библиотеку классов, а не программу, которая может быть выполнена сама по себе. (Обычно при разработке такой программы создается сопутствующий `.EXE`-проект, который предназначен для тестирования библиотеки в процессе разработки. Но когда готовая библиотека "выходит в свет", в ее состав входят только `.DLL`-файл и документация, например, сгенерированная на основании XML-комментариев. В следующем разделе показано, как написать собственную библиотеку классов.

Объединение классов в библиотеки


ЗАПОМНИ!

Простейшее определение *проекта библиотеки классов* — это классы, не содержащие метода `Main()`, что отличает библиотеку классов от выполнимой программы. Насколько верное это определение? В определенной мере — этим отличается библиотека классов от выполнимого файла. Библиотеки C# гораздо проще писать и использовать, чем аналогичные библиотеки на C или C++.

Из следующих разделов вы узнаете, как создавать собственные библиотеки классов. Не беспокойтесь, это несложно.

Создание проекта библиотеки классов

Файлы нового проекта библиотеки классов и ее драйвера можно создать любым из следующих двух способов.

- **Создайте проект библиотеки классов и добавьте к решению тестовое приложение**. Этот подход применим, если вы создаете

сборку автономной библиотеки классов. Как создается проект библиотеки классов, я расскажу в следующем разделе.

» **Создайте тестовое приложение** и **добавьте к решению один или несколько проектов библиотеки**. Таким образом, вы можете сначала создать тестовое приложение (в виде консольного приложения или графического приложения Windows Forms (или Windows Presentation Foundation)), а затем к решению добавить проекты библиотек классов.

К этому подходу можно прибегнуть, если у вас имеется приложение и вы хотите добавить к нему библиотеку поддержки. В этом случае тестовое приложение может быть либо активной программой, либо специальным проектом тестового приложения, добавленным к решению просто для тестирования библиотеки. Для тестирования проект приложения следует указать в качестве начального, как было указано ранее в этой главе.

Создание автономной библиотеки классов

Если вся ваша цель заключается в создании автономной библиотеки классов, которая может использоваться в разных программах, можете создать решение, которое содержит проект библиотеки классов, "с нуля".

1. **Выберите пункт меню Choose File⇨New⇨Project (Выберите файл⇨Новый⇨Проект).**

   Вы увидите диалоговое окно нового проекта New Project, показанное на рис. 20.1.

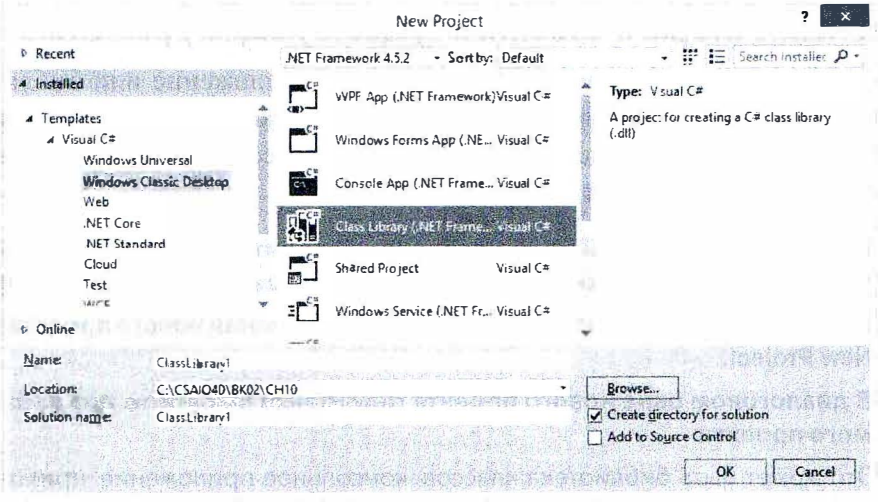


Рис. 20.1. Новая библиотека в обозревателе решений

2. **В папке Visual C#\Windows Classic Desktop выберите шаблон библиотеки классов Class Library.**

   На рис. 20.1 показан этот момент работы с обозревателем решений.

3. **Введите имя вашей библиотеки классов в поле Name и щелкните на кнопке OK.**

   В примере в качестве имени проекта используется TestClass. Visual Studio создает новый проект; после чего открывает в редакторе файл `Class1.cs`, так что вы сразу находитесь в режиме добавления в вашу библиотеку классов. После создания проекта библиотеки классов вы можете добавить проект тестового приложения (или проект модульного теста, или их оба), используя подход, описанный в следующем разделе.

СОВЕТ

В обозревателе решений вы видите, что имя файла вашего класса — `Class1.cs` — не очень информативное. Но исправить ситуацию очень легко.

1. **Щелкните правой кнопкой мыши на `Class1.cs` и выберите в контекстном меню пункт переименования Rename.**

   Теперь вы можете ввести новое имя.

2. **Введите новое имя вашего файла и нажмите клавишу <Enter>.**

   Вы увидите диалоговое окно подтверждения переименования файла.

3. **Щелкните на кнопке Yes.**

   Visual Studio автоматически переименует все ссылки на `Class1` таким образом, чтобы они соответствовали новому имени файла, введенному вами.

Добавление второго проекта к существующему решению

Если у вас есть существующее решение — приложение или библиотека классов, описанная в предыдущем разделе, — к нему легко добавить другой проект, который представляет собой библиотеку классов или выполнимое приложение, такое как тестовое приложение.

1. **При открытом в Visual Studio решении щелкните правой кнопкой мыши на узле решения (верхнем узле) в обозревателе решений Solution Explorer.**

2. **В всплывающем меню выберите пункт добавления нового проекта Add⇨ New Project.**

3. **В диалоговом окне нового проекта New Project выберите тип добавляемого проекта.**

   Это может быть библиотека классов, консольное приложение, приложение Windows Forms или любой иной доступный в правой части диалогового окна тип проекта.

4. **Воспользуйтесь полем местоположения Location, чтобы указать, где должен располагаться ваш проект.**

Папку нового проекта можно поместить в одном из двух мест.

- **В подпапке**: перейдите в папку основного проекта и добавьте подпапку дополнительного проекта в качестве подпапки (рис. 20.2).
- **На одном уровне**: перейдите в папку, в которой содержится папка основного проекта, так, чтобы оба проекта находились на одном и том же уровне (рис. 20.3).

Рис. 20.2. Размещение дополнительного проекта в подпапке основного

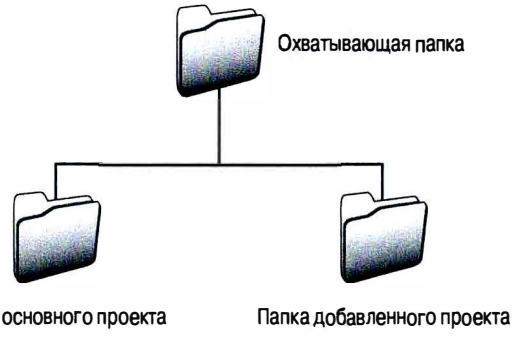


Рис. 20.3. Размещение дополнительного проекта в папке на том же уровне, что и папка основного проекта

5. **Дайте имя своему проекту и щелкните на кнопке OK.**

В нашем примере использованы имя `TestApplication` и тип консольного приложения. Если новый проект представляет собой проект библиотеки, будьте внимательны при выборе имени — оно станет именем `.DLL`-файла библиотеки и именем пространства имен, в котором содержатся классы проекта.

СОВЕТ

Если вам надо назвать проект библиотеки так же, как другой проект или даже основной проект, можете различать их, добавив суффикс `Lib` к имени проекта библиотеки, например `MyConversionLib`.

Если добавляемый вами проект предназначен для автономной работы и может использоваться в других программах, лучше использовать папки на одном уровне.

Пример `TestClass` в данном разделе (как и большинство примеров в данной книге) использует подход с вложенными подпапками. Идея в том, что папки не обязаны находиться в некотором определенном месте, но такое их размещение удобнее всего. После создания тестового приложения щелкните на нем правой кнопкой мыши в обозревателе решений и выберите из контекстного меню пункт Set as StartUp Project (Установить как начальный проект).

Выбор местоположения не зависит от того факта, что вы добавляете новый проект в решение `TestClass`. Две папки проектов могут находиться в одном решении, даже будучи размещенными в совершенно разных местах.

Создание классов для библиотеки

Сформировав проект библиотеки классов, вы создаете классы, составляющие эту библиотеку. Приведенный ниже пример `TestClass` демонстрирует простую библиотеку классов.

```
using System;
namespace TestClass
{
    public class DoMath
    {
        public int DoAdd(int Num1, int Num2)
        {
            return Num1 + Num2;
        }
        public int DoSub(int Num1, int Num2)
        {
            return Num1 - Num2;
        }
        public int DoMul(int Num1, int Num2)
        {
            return Num1 * Num2;
        }
        public int DoDiv(int Num1, int Num2)
        {
            return Num1 / Num2;
        }
    }
}
```

Библиотеки могут содержать любые типы C#: классы, структуры, делегаты, интерфейсы и перечисления. О структурах рассказывается в главе 22, "Структуры", о делегатах — в главе 19, "Делегирование событий", об интерфейсах — в главе 18, "Интерфейсы", а информацию о перечислениях можно найти в главе 10, "Списки элементов с использованием перечислений".


ЗАПОМНИ!

В коде библиотеки классов вы обычно не должны перехватывать исключения. Позвольте им добраться до кода клиента, вызывающего библиотеку. Клиент должен знать об исключении и обработать его

так, как это требуется в его приложении. Об исключениях рассказывалось в части 1, "Основы программирования на C#".

Использование тестового приложения

Сама по себе библиотека классов ничего не делает, так что нам нужно *тестовое приложение*, небольшая выполнимая программа, которая позволит тестировать работу библиотеки в процессе разработки путем вызова ее методов. Другими словами, мы пишем программу, которая использует классы и методы библиотеки. Такое поведение вы увидите позже в примере программы `TestApplication`.

ЗАПОМНИ!

Чтобы использовать свою библиотеку классов из тестового приложения, вы должны добавить ссылку на нее. Для этого щелкните правой кнопкой мыши на пункте ссылок References в разделе `Test Application` обозревателя решений и выберите Add Reference. Вы увидите диалоговое окно диспетчера ссылок Reference Manager, подобное показанному на рис. 20.4. Выберите Projects\Solution на левой панели, затем выберите `TestClass` на центральной панели и щелкните кнопкой мыши, чтобы добавить ссылку.

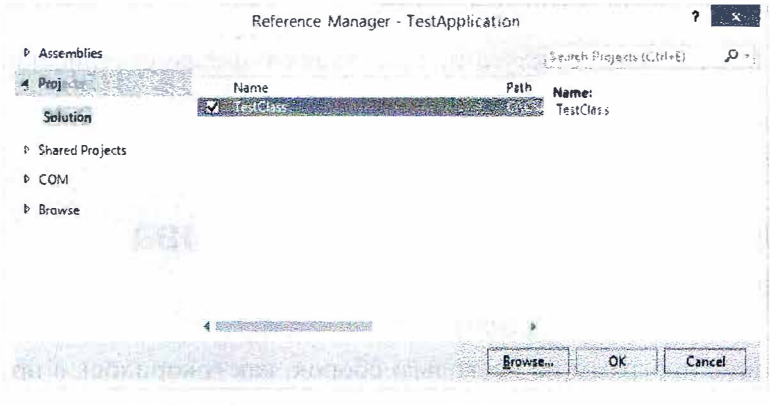


Рис. 20.4. Добавление ссылки в библиотеку классов

Приведенный далее код продолжает начатый в предыдущем разделе листинг. Это добавленный новый проект с одним классом, который содержит метод `Main()`, и в нем вы пишете код для работы с вашей библиотекой.

```
using System;
// Добавление ссылки на библиотеку классов
using TestClass;
namespace TestApplication
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            // Создание объекта DoMath
            DoMath math = new DoMath();

            // Тестирование функций DoMath
            Console.WriteLine("6 + 2 = {0}", math.DoAdd(6, 2));
            Console.WriteLine("6 - 2 = {0}", math.DoSub(6, 2));
            Console.WriteLine("6 * 2 = {0}", math.DoMul(6, 2));
            Console.WriteLine("6 / 2 = {0}", math.DoDiv(6, 2));

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}
```

Вот вывод нашего тестового приложения:

```
6 + 2 = 8
6 - 2 = 4
6 * 2 = 12
6 / 2 = 3
Нажмите <Enter> для завершения программы...
```

СОВЕТ

Библиотеки зачастую предоставляют только статические методы. В этом случае инстанцировать библиотечный объект не нужно — можно просто вызвать метод с помощью класса.

Дополнительные ключевые слова для управления доступом

Разделение программ на несколько сборок, как говорилось в предыдущих разделах, приводит к вопросу, какой код в `AssemblyB` доступен для обращения коду из `AssemblyA`. Примеры из главы 15, "Класс: каждый сам за себя", хорошо иллюстрируют применение ключевых слов `public` и `private`. Но в этой главе я ничего не рассказал вам о других ключевых словах, управляющих доступом: `protected`, `internal` и комбинации `protected internal`. В следующих разделах эта ситуация будет исправлена в предположении, что вы понимаете, что такое наследование и перекрытие методов, а также ключевого слова `public` и `private`. Чтобы этот раздел имел для вас смысл, возможно, вам следует прочитать (или перечитать) главу 15, "Класс: каждый сам за себя".

`internal`: строим глазки ЦРУ

Предположим, у нас есть программа с двумя проектами.

- Первый проект — выполнимая программа `InternalLimitsAccess`, класс которой `Congress` содержит метод `Main()`, выполняющий программу (нет такого закона, чтобы класс метода `Main()` обязательно назывался "`Program`").
- Второй проект — библиотека классов `CIAAssembly`.

В реальности Конгресс имеет раздражающую привычку лезть в дела ЦРУ и требовать от него отчета и рассказа о своих секретах — понятно, только для конгрессменов и сенаторов. "Мы никому ничего не расскажем!" Подозрительные же шпионы из ЦРУ боятся делиться своими секретами (наверняка они знают, из чего сделана кока-кола!) Допустим, что ЦРУ хотят сохранить свой самый главный секрет в полной тайне.

И тут начинаются проблемы. Все в ЦРУ должны знать этот секрет. В примере `InternalLimitsAccess` ЦРУ разделено на несколько классов, скажем, классы `GroupA` и `GroupB`. Представим, что это подразделения ЦРУ, которые иногда делятся секретами друг с другом. Предположим, что `GroupA` хранит некоторый страшный секрет X, помеченный как `private` (гриф "Перед прочтением сжечь!"). Код имеет примерно следующий вид:

```
// Сборка InternalLimitsAccess:
class Congress
{
    static void Main(...)
    {
        // Код надзора над ЦРУ
    }
}

// Сборка CIAAssembly:
public class GroupA
{
    private string _secretFormulaForCocaCola; // Секрет X
    internal GroupA() { _secretFormulaForCocaCola = "Много сахара";}
}

public class GroupB
{
    public void DoSomethingWithSecretX()
    {
        // Работаем с Secret X, если у нас есть к нему доступ
    }
}
```

Сейчас GroupB не в состоянии видеть секрет X, но этой группе требуется доступ к нему. Конечно, GroupA может объявить секрет X как имеющий статус public, но в этом случае секрет перестанет быть секретом. Если к секрету получит доступ GroupB, то точно такой же доступ получит и Congress, а также CNN, ABC и прочие телекомпании... Более того, доступ к этому секрету получит и Russia...

ЗАПОМНИ!

К счастью, на этот случай C# имеет ключевое слово internal. Это слово на один уровень ниже public, но выше private. Если пометить класс GroupA и его public-методы (т.е. видимые извне класса) ключевым словом internal, то в CIA все могут получить доступ к секрету X. Пометить можно как сам секрет, представляющий собой данные-член, так и соответствующее свойство:

```
// Сборка CIAAssembly:
internal class GroupA
{
    private string _secretFormulaForCocaCola; // Secret X
    internal string SecretX{get{return _secretFormulaForCocaCola;}}
    internal GroupA(){_secretFormulaForCocaCola = "Много сахара";}
}

public class GroupB
{
    public void DoSomethingWithSecretX()
    {
    // Работаем с Secret X
    Console.WriteLine("Я знаю секрет X длиной {0} символов,"
                      + "но не расскажу его",GroupA.SecretX.Length);
    }
}
```

Теперь класс GroupB имеет доступ к секрету, но никому о нем не рассказывает. Он может рассказать классу Congress в методе Main() о том, что он знает этот секрет и даже его длину, но сам секрет он не выдает, Congress к этому секрету обратиться не может:

```
class Congress {
    static void Main(string[] args) {
        // Код допроса CIA.
        // Следующая строка не будет скомпилирована, поскольку
        // GroupA недоступна извне сборки CIAAssembly. Congress
        // не в состоянии получить доступ к GroupA.
        // CIAAssembly.GroupA groupA = new CIAAssembly.GroupA();

        // Класс Congress может получить доступ к GroupB,
        // поскольку этот класс объявлен как public. GroupB
        // может рассказать о том, что знает секрет, но не сам
        // секрет... впрочем, тут есть небольшие хитрости.
```

```
        GroupB groupB = new GroupB();
        groupB.DoSomethingWithSecretX();

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
}
```

В методе `Main()` класс `GroupA` невидим, так что попытки создания его экземпляра не будут компилироваться. Но поскольку класс `GroupB` объявлен как `public`, метод `Main()` может обратиться к нему и вызвать его открытый метод `DoSomethingWithSecretX()`.

Но минутку! ЦРУ обязано рассказывать Конгрессу о своих секретах, пусть и ограниченному кругу лиц. Это можно организовать через тот же класс `GroupB`, как только будут представлены соответствующие полномочия (хотя их необходимо добавить в код):

```
public string
    DoSomethingWithSecretXUsingCredentials(string credentials)
{
    if (credentials == "конгрессмен с нужным допуском")
    {
        return GroupA.SecretX;
    }
    return string.Empty;
}
```

Ключевое слово `internal` делает классы и их члены доступными только из собственной сборки. В пределах сборки `internal`, по сути, представляет собой `public`.

Можно пометить метод в классе `internal` как `public`, но на самом деле он не является `public`. Член класса не может быть более доступен, чем сам класс, так что такой так называемый "`public`"-член на самом деле является `internal`.

ЦРУ все еще может хранить свои самые глубокие, самые темные секреты втайне, объявляя их `private` в своем классе. Эта стратегия делает их доступными только в этом классе.

`protected`: поделимся с подклассами

Основное назначение `private` — скрывать все, что можно. В частности, скрывать детали внутренней реализации класса. Классы, которые осведомлены о внутреннем содержании других классов, не самые везучие классы на свете.

Дело в том, что в результате они оказываются "слишком тесно связанными" с классами, о которых слишком много знают. Если класс `A` знает о том, как работает внутри класс `B`, `A` может воспользоваться этим знанием. И при малейшем изменении изменять придется оба класса.

Чем меньше другие классы и сборки знают о том, *как* класс `B` выполняет свои обязанности, тем лучше. В главе 15, "Класс: каждый сам за себя", я использовал в качестве примера класс `BankAccount`. Банк не хочет изменять мой счет непосредственно. Счет — это закрытая часть реализации класса `BankAccount`. Класс `BankAccount` предоставляет доступ к счету — но только с помощью тщательно контролируемого открытого интерфейса. В классе `BankAccount` открытый интерфейс состоит из трех открытых методов.

- Метод `Balance` предоставляет способ получить значение текущего баланса. Это свойство только для чтения, так что им нельзя воспользоваться для изменения значения баланса.
- Метод `Deposit()` позволяет строго контролируемо внести деньги на счет извне класса.
- Метод `Withdraw()` позволяет (в первую очередь, владельцу счета) снять со счета определенную сумму, но в строго контролируемых пределах. `Withdraw()` обеспечивает выполнение *бизнес-правила*, которое заключается в том, что нельзя снять со счета больше, чем на нем есть.

Здесь используются только два ключевых слова — `private` и `public`. Но в программировании бывают и иные ситуации. В предыдущем разделе вы видели, как ключевое слово `internal` открывает класс, но только другим классам в той же сборке.

Предположим, однако, что класс `BankAccount` имеет подкласс `SavingsAccount`. Методы в `SavingsAccount` требуют доступа к балансу, определенному в базовом классе. К счастью, `SavingsAccount` может использовать открытый интерфейс, как и все другие классы, — воспользоваться свойством `Balance` и методами `Deposit()` и `Withdraw()`.

Но иногда базовый класс не предоставляет такой доступ к своим внутренним делам для других. Что если член-данные `_balance` класса `BankAccount` объявлен как `private` и класс не предоставляет свойство `Balance`?


ЗАПОМНИ!

Воспользуемся ключевым словом `protected`. Если экземпляр переменной `_balance` в базовом классе объявить как `protected`, а не `private`, извне класса такая переменная будет недоступна, как и `private`. Но подклассы смогут с ней работать.

СОВЕТ

Но есть еще более корректное решение: пометить в классе BankAccount член _balance как private, как и ранее, и предоставить доступ к нему посредством свойства Balance, которое объявить как protected. Подклассы наподобие SavingsAccount могут обращаться к _balance посредством свойства Balance, но для всех внешних классов счет останется невидимым. Такой подход защищает реализацию BankAccount даже от собственных подклассов.

Если счет все же должен быть доступен (только для чтения) для внешних классов, то, конечно, следует предоставить открытое свойство Balance, которое позволяет *получить* значение счета. Если при этом вам требуется обеспечить возможность *установить* значение счета из класса SavingsAccount, вы можете обеспечить соответствующее свойство Balance с доступом protected (т.е. доступное SavingsAccount и другим подклассам, но недоступное всем остальным). В главе 15, "Класс: каждый сам за себя", рассматривалось, как это сделать:

```
// В BankAccount:
public decimal Balance
{
    get { return _balance; }              // Открытый
    protected set { _balance = value; } // Не открытый
}
```

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Подкласс BankAccount может находиться в *другой сборке*, но иметь доступ ко всему, что объявлено как protected в базовом классе BankAccount. Возможность расширения (создания подкласса) данного класса вне сборки базового класса влияет на безопасность, а потому многие классы должны быть помечены как sealed. Такое "опечатывание" класса препятствует доступу извне путем наследования. Вот почему рекомендуется делать классы расширяемыми, только если они изначально предназначены для наследования. Один из способов предоставить другому коду в той же сборке доступ к членам базового класса (включая подкласс в той же сборке) — пометить эти элементы как internal, а не как protected. Таким образом, вы получаете желаемый уровень доступа из локального подкласса, в то же время предотвращая доступ из внешнего подкласса. Конечно, доступ при этом разрешен и для других классов в сборке. Такое решение не идеально, но оно обеспечивает бо́льшую степень безопасности.

protected internal: более изощренная защита

Делая элементы базового класса BankAccount не просто protected, а protected internal, вы тем самым добавляете новое измерение элементам, доступным в вашей программе. Ключевое слово protected в одиночку

позволяет подклассам (в любой сборке программы) обращаться к помеченным как `protected` элементам базового класса. Добавление `internal` расширяет доступность элементов для всех классов, находящихся в той же сборке, что и `BankAccount`, или как минимум в подклассе в некоторой другой сборке.


ЗАПОМНИ!

Делайте элементы настолько *недоступными*, насколько это возможно. Начинайте с `private`. Если некоторые части кода требуют бо́льших прав доступа, выборочно увеличивайте их. Возможно, будет достаточно доступа `protected` (для подклассов). Если требуется доступ и для других классов в той же сборке, используйте `internal`. Если же требуется доступ как для подклассов, так и для других классов в той же сборке, применяйте `protected internal`. Модификатор `public` следует оставить только для тех классов (и их членов), доступ к которым должен быть предоставлен каждому классу программы, независимо от сборки.


СОВЕТ

Этот совет применим как к членам классов, так и к классам целиком. Делайте их как можно менее доступными. Небольшие *вспомогательные классы*, или классы, которые поддерживают реализацию некоторых более открытых классов, могут быть сделаны не более чем `internal`. Если класс или иной тип должен быть `private`, `protected` или `protected internal`, по возможности вложите его в класс, которому требуется к нему доступ.

Размещение классов в пространствах имен

Пространства имен существуют для того, чтобы можно было поместить связанные классы в "одну корзину", и для снижения коллизий между именами, используемыми в разных местах. Например, вы можете собрать все классы, связанные с математическими вычислениями, в одно пространство имен `MathRoutines`. Можно (но вряд ли практично) разделить на несколько пространств имен один исходный файл:

```
// Файл A.cs:
namespace One
{
}

namespace Two
{
}
```

Гораздо более распространена ситуация, когда несколько файлов группируются в одно пространство имен. Например, файл `Point.cs` может содержать класс `Point`, а файл `ThreeDSpace.cs` — класс `ThreeDSpace`, описывающий свойства евклидова пространства. Вы можете объединить `Point.cs`, `ThreeDSpace.cs` и другие исходные файлы C# в пространство имен `MathRoutines` (и, вероятно, в библиотечную сборку `MathRoutines`). Каждый файл будет помещать свой код в одно и то же пространство имен. (В действительности пространство имен составляют классы в этих исходных файлах, а не файлы сами по себе. В каких файлах располагаются классы, образующие пространства имен, значения не имеет. Точно так же не имеет значения и то, в каких сборках находятся эти классы — пространство имен может охватывать несколько сборок.)

```
// Файл Point.cs:
namespace MathRoutines
{
    class Point { }
}

// Файл ThreeDSpace.cs:
namespace MathRoutines
{
    class ThreeDSpace { }
}
```

Если вы не размещаете классы в пространстве имен, C# помещает их в *глобальное пространство имен*. Это базовое (безымянное) пространство имен для всех остальных пространств имен. Но все же лучше использовать конкретные пространства имен. Пространства имен служат для следующих целей.

- **Пространства имен помещают груши к грушам, а не к яблокам**. Как прикладной программист вы можете не без оснований предполагать, что все классы, составляющие пространство имен `MathRoutines`, имеют отношение к математическим вычислениям. Так что поиск некоторого математического метода следует начать с просмотра классов, составляющих пространство имен `MathRoutines`.
- **Пространства имен позволяют избежать конфликта имен**. Например, библиотека для работы с файлами может содержать класс `Convert`, который преобразует представление файла одного типа в другой. В то же время библиотека перевода может содержать класс с точно таким же именем. Назначая этим двум множествам классов пространства имен `FileIO` и `TranslationLibrary`, вы устраняете проблему: класс `FileIO.Convert`, очевидно, отличается от класса `TranslationLibrary.Convert`.

Объявление пространств имен

Пространства имен объявляются с использованием ключевого слова `namespace`, за которым следуют имя и блок в фигурных скобках. Классы в этом блоке являются частью пространства имен.

```
namespace MyStuff
{
    class MyClass {}
    class UrClass {}
}
```

В этом примере классы `MyClass` и `UrClass` являются частью пространства имен `MyStuff`.

ЗАПОМНИ!

Пространства имен неявно являются `public`, и вы не можете использовать для них никакие модификаторы (даже `public`).

Кроме классов, пространства имен могут содержать другие типы, такие как

- делегаты,
- перечисления,
- интерфейсы,
- структуры.

Пространства имен могут содержать вложенные пространства имен с любой глубиной вложенности. У вас может быть пространство имен `Namespace2`, вложенное в `Namespace1`, как показано в следующем фрагменте исходного текста:

```
namespace Namespace1
{
    // Классы в пространстве имен Namespace1...

    // Вложенное пространство имен:
    namespace Namespace2
    {
        // Классы в пространстве имен Namespace2...
        public class Class2
        {
            public void AMethod() { }
        }
    }
}
```

Для вызова метода из `Class2` в `Namespace2` откуда-то извне пространства имен `Namespace1` применяется следующая запись:

```
Namespace1.Namespace2.Class2.AMethod();
```

Рассматривайте эти пространства имен, соединенные точками, как своего рода логический путь к нужному элементу. "Имена с точками", такие как `System.IO`, выглядят, как вложенные пространства имен, но на самом деле они представляют собой имена одного пространства имен. Точно так же `System.Data` представляет собой полное имя единого пространства имен, а не имя пространства имен `Data`, вложенного в пространство имен `System`. Это соглашение упрощает возможность наличия связанных пространств имен, таких как `System.IO`, `System.Data` и `System.Text`. На практике вложенные пространства имен и пространства имен, имена которых содержат точки, неразличимы.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Удобно добавлять к пространствам имен в ваших программах название вашей фирмы: `MyCompany.MathRoutines`. (Конечно, если вы работаете на фирме; можно также использовать собственное имя.) Добавление названия фирмы предупреждает коллизии имен в вашем коде при использовании двух библиотек сторонних производителей, у которых оказывается одно и то же базовое имя пространства имен, например `MathRoutines`.

СОВЕТ

Диалоговое окно нового проекта Visual Studio **New Project** запускает мастер приложений **Application Wizard**, который помещает каждый формируемый им класс в пространство имен, имеющее такое же имя, как и создаваемый им каталог. Взгляните на любую программу в этой книге, созданную Application Wizard. Например, программа `AlignOutput` размещается в папке `AlignOutput`. Имя исходного файла — `Program.cs`, соответствующее имени класса по умолчанию. Имя пространства имен в `Program.cs` то же, что и имя папки: `AlignOutput`.

СОВЕТ

Можно изменить имя любого пространства имен, вводя новое имя. Однако, если вы не будете осторожны и аккуратны, это может привести к проблемам. Лучше щелкнуть правой кнопкой на имени пространства имен и выбрать пункт переименования **Rename** контекстного меню. Такой способ заставляет Visual Studio выполнить всю работу вместо вас и гарантировать получение корректного взаимосогласованного результата.

Пространства имен и доступ

ЗАПОМНИ!

Помимо упаковки кода в более удобный для использования вид, пространства имен расширяют понятие управления доступом, представленное в главе 15, "Класс: каждый сам за себя" (где были введены такие ключевые слова, как `public`, `private`, `protected`, `internal` и

protected internal). Пространства имен расширяют управление доступом с помощью дальнейшего ограничения на доступ к членам класса.

Однако пространства имен влияют не на *доступность*, а на *видимость*. По умолчанию классы и методы в пространстве имен NamespaceA невидимы классам в пространстве имен NamespaceB, независимо от их спецификаторов доступа. Но есть несколько способов сделать классы и методы из пространства имен NamespaceB видимыми для пространства имен NamespaceA. Обращаться вы можете только к тому, что видимо для вас.

Видимы ли вам необходимые классы и методы?

Для того чтобы определить, может ли класс Class1 в пространстве имен NamespaceA вызывать NamespaceB.Class2.AMethod(), рассмотрим следующие два вопроса.

- *Видим* ли класс Class2 из пространства имен NamespaceB вызывающему классу Class1?

  Это вопрос видимости пространства имен, который будет вскоре рассмотрен.

- Если ответ на первый вопрос — "да", то "достаточно ли открыты" Class2 и его метод AMethod() классу Class1 для доступа?

  Если Class2 находится в сборке, отличной от сборки Class1, он должен быть открыт для Class1 для доступа к его членам. Class2 в той же сборке должен быть объявлен как минимум как internal. Классы могут быть объявлены только как public, protected, internal или private.

  Аналогично метод класса Class2 должен иметь по крайней мере определенный уровень доступа в каждой из этих ситуаций. Методы добавляют protected internal в список спецификаторов доступа, имеющихся у классов. Более подробные сведения имеются в главе 15, "Класс: каждый сам за себя", и в разделе "Дополнительные ключевые слова для управления доступом" данной главы.

Для того чтобы Class1 мог вызвать метод Class2, на оба вопроса должен быть дан положительный ответ.

Как сделать видимыми классы и методы в другом пространстве имен

Язык C# предоставляет два способа для того, чтобы сделать элементы в пространстве имен NamespaceB видимыми в пространстве имен NamespaceA.

» **Применяя *полностью квалифицированные имена*** из пространства имен `NamespaceB` при использовании их в пространстве имен `NamespaceA`. Это приводит к коду наподобие приведенного, начинающемуся с имени пространства имен, к которому добавляются имя класса и имя метода:

```
System.Console.WriteLine("my string");
```

» **Устраняя необходимость в полностью квалифицированных именах** в пространстве имен `NamespaceA` посредством *директивы `using`* для пространства имен `NamespaceB`:

```
using System;  // Имена пространств имен
using NamespaceB;
```

Программы в этой книге используют последний способ — директиву `using`.

Использование полностью квалифицированных имен

Пространство имен класса является составной частью его расширенного имени, что приводит к первому способу обеспечения видимости класса из одного пространства имен в другом. Рассмотрим следующий пример, в котором нет ни одной директивы `using` для упрощения обращения к классам в других пространствах имен:

```
namespace MathRoutines  // Разбито на две части - см. ниже
{
    class Sort
    {
        public void SomeMethod(){}
    }
}
namespace Paint
{
    public class PaintColor
    {
        public PaintColor(int nRed, int nGreen, int nBlue) {}
        public void Paint() {}
        public static void StaticPaint() {}
    }
}
namespace MathRoutines  // Еще одна часть пространства имен
{
    public class Test
    {
        static public void Main(string[] args)
        {
            // Создание объекта типа Sort из того же пространства
            // имен, в котором мы находимся, и вызов некоторого
            // метода
            Sort obj = new Sort();
```

```
            obj.SomeMethod();
            // Создание объекта в другом пространстве имен –
            // обратите внимание на то, что пространство имен
            // должно быть явно включено в каждую ссылку на класс
            Paint.PaintColor black = new Paint.PaintColor(0, 0, 0);
            black.Paint();
            Paint.PaintColor.StaticPaint();
        }
    }
}
```

В обычной ситуации `Sort` и `Test` оказались бы в различных исходных файлах C#, которые вы собрали бы в одну программу. Но в этом случае классы `Sort` и `Test` содержатся внутри одного и того же пространства имен `MathRoutines`, хотя и объявляются в разных местах файла. Это пространство имен разбито на две части (в данном случае в одном и том же файле).

Метод `Test.Main()` может обращаться к классу `Sort` без указания его пространства имен, так как оба эти класса находятся в одном и том же пространстве имен. Однако метод `Main()` должен указывать пространство имен `Paint` при обращении к `PaintColor`, как это сделано в вызове `Paint.PaintColor.StaticPaint()`. Здесь использовано *полностью квалифицированное* имя.

Обратите внимание на то, что вам не требуется принимать специальных мер при обращении к `black.Paint()`, поскольку класс и пространство имен объекта `black` указаны в его объявлении.

Глава 21

Именованные и необязательные параметры

В ЭТОЙ ГЛАВЕ...

- Разница между именованными и необязательными параметрами
- Использование необязательных параметров
- Реализация ссылочных типов
- Объявление выходных параметров

Параметры, как вы, наверное, помните, являются входными данными для методов. Это значения, которые вы передаете методам, чтобы получить возвращаемое значение. Иногда — и это сбивает с толку — возвращаемые значения также являются параметрами.

В более старых версиях C# и большинстве языков, производных от C, параметры не могут быть необязательными. Вместо того, чтобы делать параметры необязательными, вы должны создавать отдельную перегрузку для каждой версии метода, который, как вы ожидаете, понадобится вашим пользователям. Эта схема вполне работоспособна, но есть некоторые проблемы, которые рассматриваются далее в этой главе. Многие программисты VB указывают на гибкую параметризацию в качестве веской причины использовать VB вместо C#.

C# версии 4.0 и выше имеют необязательные параметры. Необязательные параметры — это параметры, которые имеют значение по умолчанию прямо в сигнатуре метода, так же, как и в реализации VB.NET. Это еще один шаг во имя программирования COM. Необязательные параметры — веревка остаточной длины, чтобы на ней можно было повеситься: программист может легко ошибиться при их применении.

Изучение необязательных параметров

Необязательные параметры зависят от наличия значений по умолчанию. Например, если вы ищете номер телефона по имени и городу, то можете использовать название города по умолчанию, делая город необязательным параметром.

```
public static string searchForPhoneNumber(string name,
                                          string city = "Columbus")
{...}
```

В C# версии 3.0 (и более ранних) это можно было реализовать с помощью двух перегруженных реализаций метода поиска. Один из них включает в качестве параметров имя и город; второй — только имя. Он устанавливает значение города в теле метода и вызывает первый метод. Код при этом выглядит следующим образом:

```
public static string searchForPhoneNumber(string name,
                                          string city)
{...}

public static string searchForPhoneNumber(string name) {
    string city = "Columbus";
    return searchForPhoneNumber(name, city);
}
```

Каноническим примером является метод `addit`. Это глупо, но зато иллюстрирует реалии множественных перегрузок. Итак, у нас был такой код:

```
public static int addit(int z, int y)
{
    return z + y;
}

public static int addit(int z, int y, int x)
{
    return z+y+x;
}
```

```
public static int addit(int z, int y, int x, int w)
{
    return z + y + x + w;
}

public static int addit(int z, int y, int x, int w, int v)
{
    return z + y + x + w + v;
}
```

При наличии необязательных параметров у нас получается такой код:

```
public static int addit(int z, int y,
                        int x = 0, int w = 0, int v = 0)
{
    return z + y + x + w + v;
}
```

Если нужно сложить два параметра, это делается следующим образом:

```
int answer = addit(10, 4),
```

Если нужно сложить четыре параметра, это делается не менее легко:

```
int answer = addit(10, 4, 5, 12);
```

Чем же так опасны необязательные параметры? Дело в том, что иногда значения по умолчанию могут иметь непредвиденные последствия. Например, вы вряд ли захотите создать метод `divideit` и устанавливать значение параметра по умолчанию равным 0. Кто-то может вызвать его и получить неотлаживаемую ошибку деления на нуль. Установка дополнительных значений в методе `addit` равными 1 — тоже не слишком хорошая идея.

```
public static int addit(int z, int y, int x = 0, int w = 0, int v = 1)
{
    // Очевидно, это глупо...
    return z + y + x + w + v;
}
```

Но иногда проблемы могут быть гораздо более тонкими, поэтому используйте необязательные параметры с осторожностью. Пусть, например, у вас есть базовый класс, а затем вы порождаете класс, который реализует базовый — например, так:

```
public abstract class Base
{
    public virtual void SomeFunction(int x = 0)
    {...}
}

public sealed class Derived : Base
{
    public override void SomeFunction(int x = 1)
    {...}
}
```

Что произойдет при объявлении нового экземпляра?

```
Base ex1 = new Base();
ex1.SomeFunction();          // SomeFunction (0)
Base ex2 = new Derived();
ex2.SomeFunction();          // SomeFunction (0)
Derived ex3 = new Derived();
ex3.SomeFunction();          // SomeFunction (1)
```

Что здесь происходит? В зависимости от того, как вы реализуете классы, значение по умолчанию для необязательного параметра устанавливается по-разному. В первом примере `ex1` является объектом `Base`, и необязательный параметр по умолчанию равен `0`. Во втором примере присваивание `Derived` переменной `ex2` с применением приведения допустимо, поскольку `Derived` является подклассом `Base`, и значение по умолчанию в этом случае также равно `0`. В третьем же примере экземпляр `Derived` создается непосредственно, и значение по умолчанию равно `1`. Обычно такая разница в поведении второго и третьего примеров оказывается неожиданной для неискушенного программиста.

Ссылочные типы

Ссылочные типы, как говорилось в части 1, "Основы программирования на C#", представляют собой типы переменных, которые хранят ссылки на фактические данные вместо самих данных. Обычно о ссылочных типах говорят как об объектах. Новые ссылочные типы реализуются с помощью

- классов,
- интерфейсов и
- делегатов.

Перед тем как их использовать, их нужно создать. Сам по себе класс не является ссылочным типом, в отличие от, скажем, класса `Calendar`. Вы можете передать ссылочный тип в метод так же, как и статический тип. Он рассматривается как параметр, который вы используете внутри метода, как и любую иную переменную.

Но можно ли передавать ссылочные типы так же, как и статические типы? Давай попробуем. Например, если у вас есть метод `Schedule` в классе `Calendar`, вы можете передать в него идентификатор `CourseId` или весь `Course`. Все зависит от того, как вы структурируете приложение.

```
public class Course
{
    public int CourseId;
    public string Name;
```

```
    public void Course(int id, string name)
    {
        CourseId = id;
        Name = name;
    }
}

public class Calendar
{
    public static void Schedule(int courseId)
    {
    }
    public static void Schedule(Course course)
    {
        // Тут должно случиться что-то интересное...
    }
}
```

В этом примере у вас есть перегрузка метода `Schedule` — метод, который принимает `CourseId`, и метод, который принимает ссылочный тип `Course`. Последний тип является типом, потому что `Course` — это класс, а не статический тип, такой как `int` у `CourseId`.

Что если вы захотите, чтобы второй метод `Schedule` поддерживал необязательный параметр `Course`? Скажем, если вы хотите, чтобы он по умолчанию создавал новый курс, вы просто опускаете этот параметр? Это было бы похоже на установку статического целого числа равным `0` или какому-то иному значению, не так ли?

```
public static void Schedule(Course course = new Course())
{
    // Реализация
}
```

Однако это не разрешено. Visual Studio допускает необязательные параметры только для статических типов, и компилятор сообщит вам об этом. Вы можете обойти ограничение, принимая `CourseId` в методе `Schedule` и создавая новый курс в теле события.

Выходные параметры

Выходные параметры — это те параметры в сигнатуре метода, которые фактически изменяют значение переменной, передаваемой в них пользователем. Параметр ссылается на местоположение исходной переменной, а не создает рабочую копию. Выходные параметры объявляются в сигнатуре метода с помощью ключевого слова `out`. Таких параметров может быть столько, сколько вы захотите (в пределах разумного, конечно), хотя, если вы используете их больше, чем просто пару штук, вам, вероятно, следует задуматься об использовании

иного подхода (может быть, обобщенного списка?). Выходной параметр в объявлении метода может выглядеть следующим образом:

```
public static void Schedule(int courseId,
                            out string name,
                            out DateTime scheduledTime)
{
    name = "something";
    scheduledTime = DateTime.Now;
}
```

Следуя правилам, вы должны быть способны сделать один из этих параметров необязательным, предварительно установив его значение. В отличие от ссылочных параметров то, что выходные параметры не поддерживают значений по умолчанию, имеет смысл. Выходной параметр нужен именно для возврата значения, установка которого должна происходить внутри тела метода. Поскольку для выходных параметров какое-либо конкретное значение не ожидается, значения по умолчанию не приносят никакой выгоды программисту.

Именованные параметры

Рука об руку с концепцией необязательных параметров идет концепция именованных параметров. Если у вас есть несколько параметров по умолчанию, то нужен способ сообщить компилятору, какой именно параметр вы предоставляете методу. Например, взгляните на метод `addit`, показанный ранее в этой главе, после реализации необязательных параметров:

```
public static int addit(int z, int y, int x = 0, int w = 0, int v = 0)
{
    return z + y + x + w + v;
}
```

Очевидно, что в этой реализации порядок параметров не имеет значения, но если бы это был метод в некоторой библиотеке классов, то вы могли бы не знать, что порядок параметров не важен. Как тогда указать компилятору, что нужно пропустить параметры `x` и `w`, если вы хотите указать только `v`? В старые времена это делалось следующим образом:

```
int answer = additall(3,7,,,4);
```

К счастью, больше прибегать к этому способу нет необходимости. Теперь, при наличии именованных параметров, можно написать

```
int answer = additall(z:3, y:7, v:4);
```

Параметры, не являющиеся необязательными, не обязаны быть именованными; тем не менее использование их имен — хорошая практика. Если вы опустите их именование в приведенном примере, то получите следующий код:

```
int answer = additall(3, 7, v:4);
```

Вы должны признать, что читать такой код немного сложнее: чтобы его понять, нужно обратиться к сигнатуре метода.

Разрешение перегрузки

Проблемы начинаются, когда имеются перегруженные методы и методы с необязательными аргументами с одинаковыми сигнатурами. Поскольку C# допускает использование в перегрузках параметров с разными именами, все может быть не так уж страшно.

Рассмотрим код

```
class Course
{
    public void New(object course)
    {
    }
    public void New(int courseId)
    {
    }
}
```

Попробуем вызвать метод `New` следующим образом:

```
Course course = new Course();
course.New(10);
```

Здесь выбирается вторая перегрузка метода, потому что `10` лучше соответствует `int`, чем `object`. То же самое верно и при работе с перегруженными сигнатурами методов с необязательными параметрами — выбирается перегрузка с наименьшим количеством приведений типов, необходимых для ее работы.

Альтернативные методы возврата значений

C# 7.0 изменяет способ возврата значений. Теперь вы можете работать со ссылочными переменными и переменными `out` по-новому. В следующих разделах обсуждаются эти новые методы.

Работа с переменными out

Выше, в разделе "Выходные параметры", рассматриваются общие методы работы с выходными переменными. Давайте рассмотрим следующий пример, в котором есть только одна out-переменная.

```
static void MyCalc(out int x)
{
    x = 2 + 2;
}
```

В этом случае мы можем вызывать метод MyCalc() старым способом:

```
static void DisplayMyCalc()
{
    int p;
    MyCalc(out p);
    Console.WriteLine($"{nameof(p)} = {p}");
}
```

Вывод DisplayMyCalc() имеет вид

```
p = 4
```

Обратите внимание, что метод MyCalc() присваивает значение 4 переменной p. C# предоставляет возможность записать то же самое более кратко:

```
static void DisplayMyCalc()
{
    MyCalc(out int p);
    Console.WriteLine($"{nameof(p)} = {p}");
}
```

Вывод получается таким же, как и раньше. Однако теперь вам не нужно объявлять переменную p перед ее использованием. Объявление является частью вызова MyCalc().

СОВЕТ

Конечно, если ваш метод возвращает только один выходной параметр, обычно лучше использовать вместо него возвращаемое значение. В этом примере использован только один параметр, просто чтобы вы лучше поняли, как работает новая техника.

ЗАПОМНИ!

Более интересным дополнением к C #7.0 является то, что теперь вы можете использовать ключевое слово var с параметрами out. Например, следующий вызов вполне корректен в C# 7.0.

```
static void DisplayMyCalc()
{
    MyCalc(out var p);
    Console.WriteLine($"{nameof(p)} = {p}");
}
```

Возврат значений по ссылке

В более старых версиях C# можно возвращать значения по ссылке. Однако вы должны быть очень внимательными при написании кода:

```
static ref int ReturnByReference()
{
    int[] arrayData = { 1, 2 };
    ref int x = ref arrayData[0];
    return ref x;
}
```

В C# 7.0 можно уменьшить размер необходимого кода:

```
static ref int ReturnByReference()
{
    int[] arrayData = { 1, 2 };
    return ref arrayData;
}
```

ЗАПОМНИ!

Однако обратите внимание, что теперь вместо одного значения типа `int` вы возвращаете весь массив. Массив является ссылочным типом; `int` является типом-значением. С помощью этой методики вы не можете возвращать типы-значения. Чтобы сделать это возможным, требуется передать его как параметр, например:

```
static ref int ReturnByReference(ref int myInt)
{
    myInt = 1;
    return ref myInt;
}
```

Глава 22

Структуры

В ЭТОЙ ГЛАВЕ...

- Когда следует использовать структуры
- Определение структур
- Работа со структурами

Структуры являются важным дополнением к C#, поскольку они предоставляют средства для определения сложных объектов данных, схожих с записями баз данных. Из-за способа использования структур при разработке приложений структуры и классы во многом перекрываются. Такое перекрытие вызывает проблемы у многих разработчиков, поскольку может оказаться трудно определить, когда использовать структуру, а когда — класс. Следовательно, в первую очередь, в этой главе обсуждаются различия между структурами и классами и предлагаются некоторые лучшие практики их применения.

Создание структур требует использования ключевого слова `struct`. Структура может содержать множество тех же элементов, что и классы: конструкторы, константы, поля, методы, свойства, индексаторы, операторы, события и даже вложенные типы. Эта глава поможет вам понять тонкости создания структур с данными элементами, чтобы вы могли получить полный доступ ко всей гибкости, которую могут предложить структуры.

ЗАПОМНИ!

Несмотря на то что структуры обладают достаточно большим количеством возможных применений, наиболее распространенный способ их использования состоит в представлении записей данных. В последнем разделе этой главы рассматривается структура как объект для хранения записей. Вы узнаете, как использовать структуры для хранения отдельных записей и для ряда записей как части коллекции.

Сравнение структур и классов

Многих разработчиков различия между структурами и классами сбивают с толку, если не сказать больше. Фактически многие разработчики используют только лишь классы и забывают о структурах. Однако отказ от использования структур является ошибкой, поскольку они предназначены для выполнения определенных задач в программировании. Использование структур может сделать приложение, которое корректно выполняет свою работу, но делает это медленнее, чем могло бы, не только корректным, но и эффективным.

ЗАПОМНИ!

Имеется много подходов к использованию структур в программировании. В этой книге мы даже не будем пытаться охватить их все. В лучшем случае это краткий обзор того, как структуры могут помочь создавать лучшие приложения. Переварив полученную информацию, вы сможете начать использовать структуры и понять, как именно вы хотите с ними работать в дальнейшем.

Ограничения структур

Структуры являются типами-значениями, а это означает, что C# выделяет память для них не так, как для классов. Большинство ограничений структур связаны с этим различием. Вот некоторые соображения, которые следует учитывать, раздумывая об использовании структуры вместо класса.

- Структуры могут иметь конструкторы, но не деструкторы. Это означает, что вы можете выполнять все обычные задания, необходимые для создания определенного типа данных, но не имеете контроля над очисткой с помощью деструктора.
- Структуры не могут наследовать другие структуры и классы.
- Структуры могут реализовывать один или несколько интерфейсов, но с ограничениями, накладываемыми элементами, которые они поддерживают (подробности см. в разделе "Добавление распространенных элементов структур" далее в этой главе).
- Структуры не могут быть определены как `abstract`, `virtual` или `protected`.

Различия типов-значений

При работе со структурами вы должны помнить, что это тип-значение, а не ссылочный тип, такой как классы. Это означает, что структуры имеют определенные преимущества по сравнению с классами. Например, они гораздо менее ресурсоемки. Кроме того, поскольку структуры не собираются сборщиком мусора, им обычно требуется меньше времени для выделения и освобождения памяти.

СОВЕТ

Различия в использовании ресурсов, а также во времени выделения и освобождения памяти только усугубляются при работе с массивами. Массив ссылочных типов влечет за собой огромные накладные расходы, поскольку содержит только указатели на отдельные объекты. Чтобы получить доступ к объекту, приложение должно найти его в куче.

Типы значений являются детерминированными. Вы знаете, что C# освобождает их в тот момент, когда они выходят из области видимости. Ожидание, когда C# выполнит сборку мусора для ссылочных типов означает, что вы не можете быть полностью уверены, как именно в вашем приложении используется память.

Когда следует использовать структуры

Существует множество мнений о том, когда лучше использовать структуру, а не класс. По большей части все зависит от того, чего именно вы пытаетесь достичь, и от того, чем вы готовы платить за использование ресурсов и скорость приложения. В большинстве случаев классы используются гораздо чаще структур просто потому, что классы более гибки и в некоторых ситуациях имеют меньшие накладные расходы.

Как и все типы значений, структуры должны быть упакованы и распакованы при приведении к ссылочному типу или когда это требуется интерфейсом, который они реализуют. Слишком много упаковок и распаковок на самом деле заставят ваше приложение работать медленнее. Это означает, что, когда вам нужно выполнять задачи со ссылочными типами, следует избегать использования структур. В этом случае лучше всего использовать класс.

ЗАПОМНИ!

Использование типа-значения также изменяет способ взаимодействия C# с переменной. Ссылочный тип передается во время вызова по ссылке, поэтому любые изменения, внесенные в ссылочный тип, появляются в экземпляре, на который указывает эта ссылка. Тип-значение передается по значению и копируется при передаче. Это означает, что изменения, которые вы вносите в тип-значение в методе,

не отображаются в исходной переменной. Возможно, это наиболее запутанный аспект использования структур для разработчиков, поскольку передача объекта, созданного классом, по своей сути отличается от передачи переменной, созданной структурой. Это различие делает классы в целом более эффективными, чем структуры, для передачи в методы.

Структуры имеют определенное преимущество при работе с массивами. Однако вы должны проявлять осторожность при работе со структурами в типах коллекций, потому что структура может требовать упаковку и распаковку. Если коллекция работает с объектами, следует рассмотреть возможность использования класса, а не структуры.

Избегайте использования структур при работе с объектами. Да, можно размещать типы объектов внутри структуры, но тогда структура будет содержать не сам объект, а ссылку на него. По возможности ограничивайте структуры применением других типов-значений, таких как `int` и `double`. Конечно, многие структуры все равно используют ссылочные типы, такие как `String`.

Создание структур

Создание структуры во многом похоже на создание класса. Конечно, вы используете ключевое слово `struct` вместо ключевого слова `class`, а сама структура имеет ограничения, описанные выше, в разделе "Ограничения структур". Однако даже с учетом этих различий, если вы знаете, как создать класс, вы можете создать и структуру. В следующих разделах более подробно описывается, как работать со структурами.

Определение базовой структуры

Базовая структура не содержит ничего, кроме полей для хранения данных. Например, рассмотрим структуру для хранения сообщений от людей, запрашивающих цену определенного товара для некоторого его количества. Она может иметь следующий вид:

```
public struct Message
{
    public int MsgID;
    public int ProductID;
    public int Qty;
    public double Price;
}
```

Для использования такой базовой структуры можно следовать схеме наподобие следующей:

```
// Создание структуры
Message myMsg = new Message();

// Создание сообщения
myMsg.MsgID = 1;
myMsg.ProductID = 22;
myMsg.Qty = 5;

// Вычисление цены
myMsg.Price = 5.99 * myMsg.Qty;

// Вывод структуры на экран
Console.WriteLine("Сообщаем в ответ на сообщение {0},"+
                  " что вы можете получить {1} единиц товара {2}"+
                  " на общую сумму {3}.",
                  myMsg.MsgID, myMsg.Qty,
                  myMsg.ProductID, myMsg.Price);
```

Обратите внимание, что процесс создания и использования структуры очень похож на процесс создания и использования класса. Фактически их можно рассматривать как в основном одинаковые (не забывая о том, что в действительности структуры отличаются от классов). Вывод этого фрагмента кода выглядит следующим образом:

```
Сообщаем в ответ на сообщение 1, что вы можете получить
    ↳ 5 единиц товара 22 на общую сумму 29.95.
```

ЗАПОМНИ!

Очевидно, что это упрощенный пример, и вы никогда не будете создавать подобный код для реального приложения, но он демонстрирует схему использования структур. Работая со структурами, думайте о схеме работы с классами, но с некоторыми отличиями, которые могут сделать структуры более эффективными в использовании.

Добавление распространенных элементов структур

Структуры могут включать в себя многие из элементов, которые включают классы. Раздел "Определение базовой структуры" данной главы знакомит вас с использованием полей. Как отмечалось ранее, поля не могут быть объявлены как `abstract`, `virtual` или `protected`. Тем не менее их область видимости по умолчанию является `private`, но ее можно сделать `public`, как показано в коде. Очевидно, что классы содержат гораздо больше, чем только поля, и это справедливо и в отношении структур. В следующих разделах вы познакомитесь с распространенными элементами структур, чтобы уметь эффективно использовать структуры в своем коде.

Конструкторы

Как и в случае класса, вы можете создать структуру с конструктором. Вот пример `struct Message` с конструктором:

```
public struct Message
{
    public int MsgID;
    public int ProductID;
    public int Qty;
    public double Price;

    public Message(int msgId, int productId = 22, int qty = 5)
    {
        // Предоставляется пользователем
        MsgID = msgId;
        ProductID = productId;
        Qty = qty;

        // Определяется приложением
        if (ProductID == 22)
        {
            Price = 5.99 * qty;
        }
        else
        {
            Price = 6.99 * qty;
        }
    }
}
```


ЗАПОМНИ!

Обратите внимание, что конструктор принимает значения параметров по умолчанию, поэтому вы можете использовать один конструктор несколькими способами. Когда вы используете новую версию `Message`, IntelliSense показывает вам как конструктор по умолчанию (который, в отличие от класса, не исчезает при создании пользовательского конструктора), так и новый конструктор, который вы создали:

```
// Создание структуры с использованием конструктора
Message myMsg2 = new Message(2);

// Вывод структуры на экран
Console.WriteLine("Сообщаем в ответ на сообщение {0},"+
                  " что вы можете получить {1} единиц товара {2}"+
                  " на общую сумму {3}.",
                  myMsg.MsgID, myMsg.Qty,
                  myMsg.ProductID, myMsg.Price);
```

Благодаря наличию параметров со значениями по умолчанию вы можете создать новое сообщение, просто указав его номер. Параметры со значениями

по умолчанию присвоят значения другим полям. Конечно, вы можете переопределить любое из значений, чтобы создать уникальный объект.

Константы

Как и во всех других областях C#, вы можете определять в структурах константы, которые служат удобочитаемыми формами не изменяющихся значений. Например, вы можете создать константу обобщенного продукта следующим образом:

```
public const int genericProduct = 22;
```

Конструктор `Message` после этого может принять следующий вид:

```
public Message(int msgId,
               int productId = genericProduct,
               int qty = 5)
```

Новый вид конструктора удобнее для чтения, но имеет тот же результат работы.

Методы

Структуры часто могут выиграть от добавления методов, которые помогут выполнять конкретные задачи с этими структурами. Например, вы можете захотеть предоставить метод для вычисления поля `Price` (цена), а не вычислять его всякий раз вручную. Использование метода гарантирует, что изменение в методе расчета появляется в вашем коде только один раз, а не каждый раз, когда приложению требуется вычисленное значение. Метод `CalculatePrice()` может выглядеть следующим образом:

```
public static double CalculatePrice(double SinglePrice, int Qty)
{
    return SinglePrice * Qty;
}
```

Очевидно, что большинство расчетов не так просты, но идея должна быть понятна. Перемещение кода в метод означает, что вы можете изменить другие части кода, сделав его более понятным. Например, конструкция `if` в конструкторе `Message()` теперь выглядит так:

```
// Определяется приложением
if (ProductID == 22)
{
    Price = CalculatePrice(5.99, qty);
}
else
{
    Price = CalculatePrice(6.99, qty);
}
```

ЗАПОМНИ!

Обратите внимание, что вы должны объявить метод как `static`, иначе вы получите сообщение об ошибке. Структура, как и класс, может иметь методы как структуры, так и экземпляра. Методы экземпляра становятся доступными только после создания экземпляра структуры.

Свойства

Вы также можете использовать со структурами свойства. Фактически использование свойств является рекомендуемым подходом во многих случаях, потому что оно позволяет гарантировать корректность входных значений. К счастью, если вы используете C# 7.0 и изначально создали открытые поля, вы можете легко превратить их в свойства, выполнив следующие действия.

1. **Поместите курсор (точку ввода) в любом месте строки кода, который вы хотите превратить в свойство.**

   В данном случае поместите его где-нибудь в строке кода, которая гласит `public int MsgID;`. В левом поле области редактирования появится пиктограмма лампочки.

2. **Установите указатель мыши на лампочку, чтобы отобразить направленную вниз стрелку рядом с пиктограммой, и щелкните на этой стрелке.**

   Вы увидите варианты выбора, показанные на рис. 22.1. Выделенный вариант, Encapsulate Field: 'MsgID' (and use property) (Инкапсулировать поле `MsgID` (и использовать свойство)), позволяет превратить `MsgID` в свойство и использовать его в своем коде соответствующим образом.

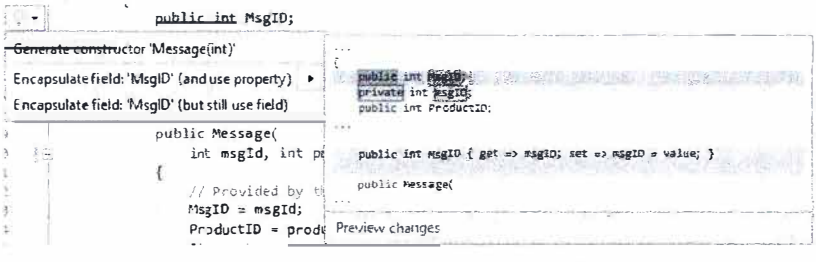


Рис. 22.1. Превращение поля `MsgID` в свойство `MsgID`

3. **Щелкните на подсвеченном на рис. 22.1 пункте.**

   Visual Studio превратит это поле в свойство, внося изменения, выделенные полужирным шрифтом в приведенном далее фрагменте кода:

```
private int msgID;
public int ProductID;
public int Qty;
public double Price;
public const int genericProduct = 22;
public int MsgID { get => msgID; set => msgID = value; }
```

СОВЕТ

На этом этапе при необходимости защиты данных вы можете работать со свойством. Однако есть еще одна проблема. Если вы попытаетесь скомпилировать свой код, то увидите в конструкторе сообщение об ошибке CS0188, гласящее, что вы пытаетесь использовать свойство до присваивания полей. Чтобы устранить эту проблему, замените присваивание `MsgID = msgId;` в конструкторе присваиванием `msgID = msgId;`. Разница в том, что теперь выполняется присваивание значение закрытому полю, а не используется открытое свойство.

Индексаторы

Индексатор позволяет работать со структурой как с массивом. Фактически при работе со структурами можно использовать многие методики для работы с массивами, но кроме того, можно создать множество новых функциональных возможностей "с нуля", потому что индексатор структуры обладает гибкостью, которую не обеспечивает массив. Вот код структуры `ColorList`, которая обеспечивает основные возможности, необходимые для индексатора:

```
public struct ColorList
{
    private string[] names;

    public string this[int i]
    { get => names[i]; set => names[i] = value; }

    public void Add(string ColorName)
    {
        if (names == null)
        {
            names = new string[1];
            names[0] = ColorName;
        }
        else
        {
            names = names.Concat<string>(
                    new string[] { ColorName }).ToArray();
        }
    }

    public int Length
    { get => names.Length; }
}
```

ЗАПОМНИ!

В верхней части листинга указывается наличие в структуре массива, в данном случае — массива `names`. Чтобы получить доступ к `names` с помощью индексатора, вы также должны создать свойство `this` типа, показанного в примере. Это свойство позволяет получить доступ к определенным элементам массива имен. Обратите внимание,

что в данном примере используется очень простое свойство this; в производственной версии в него были бы добавлены все виды проверок, включая проверку того, что names не равен null и что запрошенное значение действительно существует.

При работе с индексатором, связанным с классом, вы присваиваете массиву начальное значение. Однако в данном случае вы не можете это сделать, потому что это структура; поэтому names остается неинициализированным. Другая проблема заключается в том, что вы не можете перекрыть конструктор по умолчанию, поэтому не можете инициализировать names в нем. Решение заключается в методе Add(). Чтобы добавить новый член к names, вызывающий метод должен предоставить строку, добавляемую в names, как показано в коде выше.

Обратите внимание, что когда names имеет значение null, метод Add() сначала инициализирует массив, а уже затем добавляет цвет к первому элементу (учитывая, что других элементов нет). Однако, когда names уже содержит значения, код добавляет новый одноэлементный массив строк к names. Для преобразования перечислимого типа, используемого с Concat(), в массив для последующего сохранения в names следует вызвать ToArray().

Чтобы использовать ColorList в реальном приложении, необходимо предоставить средство получения длины массива. Свойство Length (только для чтения) выполняет эту задачу, предоставляя значение names.Length. Вот пример ColorList в действии:

```
// Создание списка цветов
ColorList myList = new ColorList();

// Заполнение его значениями
myList.Add("Yellow");
myList.Add("Blue");

// Поочередный вывод всех элементов
for(int i = 0; i < myList.Length; i++)
    Console.WriteLine("Color = " + myList[i]);
```

Код работает так, как и следовало ожидать для пользовательского массива. Вы создаете новый ColorList, с помощью Add() добавляете к нему значения, а затем используете Length в цикле for для отображения значений. Вот вывод этого кода:

```
Color = Yellow
Color = Blue
```

Операторы

Структуры могут также содержать операторы. Например, вы можете создать метод для сложения двух структур ColorList. Вы делаете это, создавая

оператор +. Обратите внимание, что вы не переопределяете оператор +, а создаете его, как показано далее:

```
public static ColorList operator+(ColorList First, ColorList Second)
{
    ColorList Output = new ColorList();
    for (int i = 0; i < First.Length; i++)
        Output.Add(First[i]);
    for (int i = 0; i < Second.Length; i++)
        Output.Add(Second[i]);
    return Output;
}
```

Вы не можете создать оператор экземпляра. Он, как показано в приведенном коде, должен быть частью структуры. Процесс следует той же методике, что и используемая для создания `ColorList`. Разница в том, что для решения задачи выполняется итерирование обеих переменных `ColorList` с использованием цикла `for`. Вот пример кода, который использует оператор + для сложения двух переменных `ColorList`.

```
// Создание и заполнение второго списка цветов
ColorList myList2 = new ColorList();
myList2.Add("Red");
myList2.Add("Purple");

// Добавление первого списка ко второму
ColorList myList3 = myList + myList2;

// Поочередный вывод всех элементов
for (int i = 0; i < myList3.Length; i++)
    Console.WriteLine("myList3 Color = " + myList3[i]);
```

Как видите, `myList3` является результатом сложения двух переменных `ColorList`, а не создания новой. Результат выглядит именно так, как и ожидается:

```
myList3 Color = Yellow
myList3 Color = Blue
myList3 Color = Red
myList3 Color = Purple
```

Использование структур как записей

В большинстве случаев основной причиной работы со структурами является создание записей, содержащих пользовательские данные. Эти пользовательские записи данных используются для хранения сложной информации и передачи ее по мере необходимости в различные методы. Проще и быстрее передать одну запись, чем целый набор значений данных, особенно когда ваше

приложение выполняет такие действия постоянно. В следующих разделах показано, как использовать структуры в качестве разновидности записей данных.

Управление отдельной записью

Передача структур в методы является более понятной и простой, чем передача набора отдельных значений данных. Конечно, чтобы эта стратегия корректно работала, значения в структуре должны быть взаимосвязаны. Рассмотрим следующий метод:

```
static void DisplayMessage(Message msg)
{
Console.WriteLine("Сообщаем в ответ на сообщение {0},"+
                  " что вы можете получить {1} единиц товара {2}"+
                  " на общую сумму {3}.",
                  msg.MsgID, msg.Qty,
                  msg.ProductID, msg.Price);
}
```

СОВЕТ

Здесь метод `DisplayMessage()` получает единственный входной аргумент типа `Message` вместо четырех переменных, которые требовались бы для такого метода без применения структур. Использование структуры `Message` приводит к следующим положительным результатам.

- Метод-получатель может считать, что в наличии имеются все необходимые значения данных.
- Метод-получатель может считать, что все переменные инициализированы.
- Вызывающий код менее склонен к ошибкам.
- Код гораздо проще для понимания другими программистами.
- В такой код легче вносить изменения.

Добавление структур в массивы

Приложения редко используют единственную запись данных для всех целей. В большинстве случаев приложения также включают коллекции записей, подобные базам данных. Например, приложение вряд ли получит только одно сообщение. Скорее всего, оно получит целую группу записей `Message`, каждая из которых должна быть обработана.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Структуры можно добавлять в любую коллекцию. Однако большинство коллекций работают с объектами, поэтому добавление к ним структур повлечет за собой снижение производительности из-за упаковок и распаковок каждой структуры. По мере увеличения

размера коллекции накладные расходы на упаковку и распаковку становятся весьма заметными. Поэтому, когда скорость является наиболее важной целью, а приложение работает только с записями данных, представленными структурами, лучше всего ограничиться массивами.

Работа с массивом структур очень похожа на работу с массивом чего-либо иного. Вы можете использовать для создания массива структур `Message` код, подобный показанному далее:

```
// Вывод всех сообщений на экран
Message[] Msgs = { myMsg, myMsg2 };
DisplayMessages(Msgs);
```

В этом случае `Msgs` содержит две записи, `myMsg` и `myMsg2`. Затем код обрабатывает сообщения, передавая массив показанному ниже методу `DisplayMessages()`:

```
static void DisplayMessages(Message[] msgs)
{
    foreach (Message item in msgs)
    {
        Console.WriteLine("Сообщаем в ответ на сообщение {0},"+
              " что вы можете получить {1} единиц товара {2}"+
              " на общую сумму {3}.",
              msg.MsgID, msg.Qty,
              msg.ProductID, msg.Price);
    }
}
```

Метод `DisplayMessages()` использует для разделения отдельных записей `Message` цикл `foreach`. Затем он обрабатывает их, используя тот же подход, что и метод `DisplayMessage()` из предыдущего раздела главы.

Перекрытие методов

ЗАПОМНИ!

Структуры обеспечивают большую гибкость, которую многие разработчики предполагают присущей исключительно классам. Например, вы можете перекрывать методы, зачастую такими способами, которые делают вывод структур гораздо лучшим. Хорошим примером является метод `ToString()`, который выводит нечто бесполезное, подобное показанной ниже строке:

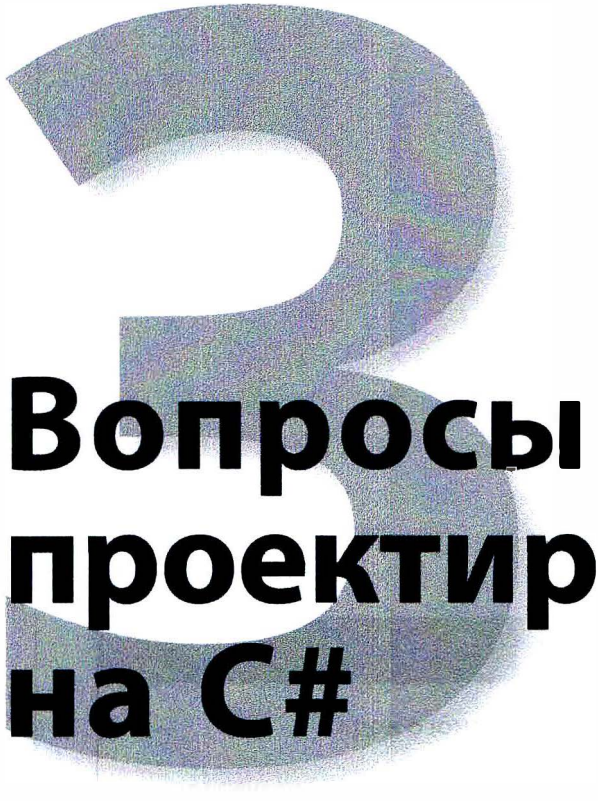
```
Structures.Program+Messages
```

Этот вывод бесполезен, потому что ничего не говорит пользователю. Чтобы получить нужную информацию, следует переопределить метод `ToString()` с помощью кода наподобие следующего:

```
public override string ToString()
{
    // Создание информативной строки
    return "Message ID:\t" + MsgID +
           "\r\nProduct ID:\t" + ProductID +
           "\r\nQuantity:\t" + Qty +
           "\r\nTotal Price:\t" + Price;
}
```

Теперь при вызове `ToString()` вы получаете полезную информацию, например при вызове `myMsg.ToString()` будет получен следующий вывод:

```
Message ID: 1
Product ID: 22
Quantity: 5
Total Price: 29.95
```



Вопросы проектирования на C#

В ЭТОЙ ЧАСТИ...

Глава 23

Написание безопасного кода

В ЭТОЙ ГЛАВЕ...

- Проектирование безопасности
- Создание безопасных приложений для Windows и веба
- Использование `System.Security`

Безопасность — это большая тема. Если проигнорировать все модные слова, связанные с безопасностью, то вы, вероятно, и сами понимаете, что вам нужно защитить свое приложение от использования людьми, которые не должны его использовать. Вам необходимо также запретить использование своего приложения для того, для чего оно использоваться не должно.

В начале электронной эры безопасность обычно осуществлялась путем *запутывания* (obfuscation). Если у вас было приложение, в которое не должны были заглядывать посторонние, вы просто прятали его, и никто не знал, где его найти. (Вспомните фильм *Военные игры*, в котором военные предполагали, что никто не в состоянии найти телефонный номер, необходимый для подключения к их мэйнфреймам, но персонаж Мэтью Бродерика все равно это сделал.)

Очевидно, что в нынешнем взрослом мире игра в прятки — это просто несерьезно. Сейчас нужно рассматривать безопасность как неотъемлемое требование каждой системы, которую вы пишете. Ваше приложение может не содержать конфиденциальных данных, но не может ли оно использоваться для

получения другой информации с компьютера? Нельзя ли использовать его для получения доступа к сети, доступ к которой предоставляться не должен? Ответы на эти вопросы имеют важное значение.

Двумя основными частями безопасности являются аутентификация и авторизация. *Аутентификация* (authentication) — это процесс проверки подлинности пользователя (действительно ли он тот, за кого себя выдает). Наиболее распространенный метод аутентификации — использование имени пользователя и пароля, хотя существуют и другие способы, такие как сканирование отпечатков пальцев. *Авторизация* (authorization) — это действие, гарантирующее, что пользователь имеет полномочия для выполнения определенных задач. Хорошим примером являются права доступа к файлам, например пользователи не могут удалять системные файлы.

ВНИМАНИЕ!

Невозможно идентифицировать конкретного пользователя с полной уверенностью. Хакеры могут легко украсть имена пользователей и пароли. Биометрические устройства, такие как сканеры отпечатков пальцев, также чрезвычайно просто обмануть. В статье по адресу `http://www.instructables.com/id/How-To-Fool-a-Fingerprint-Security-System-As-Easy-/` подробно рассказывается, как преодолеть безопасность на основе отпечатков пальцев. Лучшее, на что вы можете надеяться, — это аутентифицировать учетные данные, а не самого пользователя. Вы не в состоянии узнать, что имеете дело с конкретным человеком; это может быть всего лишь замаскировавшийся хакер.

Современные системы безопасности усложняют введение вашей системы в заблуждение о подлинности учетных данных пользователя или его авторизации. Требования безопасности — это нечто большее, чем просто добавление в вашу программу текстовых полей для имени пользователя и пароля. В этой главе вы познакомитесь с инструментами, доступными в .NET Framework, которые помогут вам обеспечить безопасность ваших приложений.

Проектирование безопасного программного обеспечения

Безопасность требует значительного количества работы по точному проектированию. Разбив процесс на части, вы обнаружите, что его гораздо проще выполнить. Команда Patterns and Practices (группа разработчиков программного обеспечения в Microsoft, разрабатывающих лучшие практики программирова-

ния) предложила системный подход к разработке безопасных программ, описанный в следующих разделах.

Определение того, что следует защищать

В разных приложениях в защите нуждаются разные артефакты, но все приложения имеют что-то, что требуется защищать. Если в вашем приложении есть база данных, это самый важный элемент для защиты. Если ваше приложение является серверным приложением, то при определении того, что именно следует защищать, сервер должен иметь весьма высокую оценку.


ЗАПОМНИ!

Даже если ваша программа представляет собой небольшое однопользовательское приложение, программа не должна делать что-то неверное. Посторонний не должен иметь возможность использовать ваше приложение для взлома компьютера пользователя.

Документирование компонентов программы

Если вы думаете, что название этого раздела звучит похоже на часть процесса проектирования, вы правы. Большая часть моделирования угроз состоит в простом понимании, как работает приложение, и тщательном его описании.

Сначала опишите, что делает приложение. Это описание становится функциональным обзором. Если вы следуете общепринятому жизненному циклу разработки программного обеспечения (Software Development Life Cycle — SDLC), то хорошей отправной точкой являются примеры использования, требования или пользовательские истории (в зависимости от вашей личной методологии).

Далее опишите, как именно приложение выполняет свои задачи на самом высоком уровне. В этом вам поможет диаграмма обзора архитектуры программного обеспечения (Software Architecture Overview — SAO). Эта диаграмма показывает, какие машины и сервисы делают то или иное в вашем программном обеспечении. Иногда SAO представляет собой простую диаграмму. Если у вас есть автономная программа Windows Forms (также известная как WinForms), такая как игра, этого достаточно. Автономная программа не имеет сетевых подключений и связи между частями программного обеспечения. Следовательно, диаграмма архитектуры программного обеспечения содержит только один объект.

Разложение компонентов на функции

После создания документа, в котором описывается, что и как делает ваше программное обеспечение, необходимо выделить его отдельные

функциональные части. Если вы создавали свою программу из компонентов, то классы и методы демонстрируют его функциональную декомпозицию. Это проще, чем кажется.

Конечным результатом разбиения программного обеспечения на отдельные части является выяснение того, какие компоненты должны быть защищены, какие части программного обеспечения взаимодействуют с каждым из компонентов, какие части сети и аппаратной системы взаимодействуют с компонентами и какие функции программы выполняет каждый компонент.

Обнаружение потенциальных угроз в функциях

Создав список компонентов, которые нужно защитить, вы решите самую сложную часть проблемы. Выявление угроз — это процесс, который приносит консультантам по безопасности большие деньги и почти полностью зависит от опыта.

Например, если ваше приложение подключается к базе данных, вы должны представить, что подключение может быть перехвачено третьей стороной. Если вы используете для хранения конфиденциальной информации файл, теоретически он может быть скомпрометирован.

Чтобы создать модель угроз, нужно классифицировать потенциальные угрозы для своего программного обеспечения, возможная разбивка которых на категории приведена далее.

- **Подмена учетных данных**: пользователи притворяются кем-то, кем на самом деле не являются.
- **Подделка данных или файлов**: пользователи редактируют что-то, что не должно быть изменено.
- **Отказ от действий**: пользователи имеют возможность заявить, что они не делали то, что на самом деле делали.
- **Раскрытие информации**: пользователи видят то, чего не должны видеть.
- **Отказ в обслуживании**: пользователи запрещают доступ к системе законным пользователям.
- **Повышение привилегий**: пользователи получают доступ к тому, к чему у них не должно быть доступа.

Все эти угрозы должны быть в общих чертах задокументированы под функциями, которые представляют угрозу. Такая стратегия не только дает хороший, дискретный список угроз, но и фокусирует усилия по защите на тех частях приложения, которые представляют наибольшую угрозу для безопасности.

Оценка рисков

Последний шаг в этом процессе заключается в оценке рисков. Microsoft использует пять ключевых атрибутов, используемых для измерения каждой уязвимости.

- **Потенциальный ущерб**: в какую сумму обойдется компании эта брешь в защите.
- **Воспроизводимость**: особые условия для брешей, которые могут затруднить или облегчить их поиск.
- **Уязвимость**: мера того, как далеко хакер может проникнуть в корпоративную систему.
- **Затронутые пользователи**: количество пользователей, которые могут быть затронуты проблемами, и кто они.
- **Обнаруживаемость**: легкость, с которой можно найти потенциальное нарушение.

Вы можете прочесть об этой модели по адресу `http://msdn.microsoft.com/security` или использовать собственную модель угроз для учета указанных атрибутов. Ключевая задача состоит в том, чтобы определить, какие угрозы могут вызвать проблемы, а затем смягчить их.

Построение безопасных приложений Windows

При работе на клиентском компьютере каркас .NET находится в жестко контролируемой изолированной программной среде ("песочнице"). Реалии этой изолированной среды приводят к особой важности настройки стратегии безопасности для вашего приложения.

Первое, на что вам нужно обратить внимание при обеспечении безопасности в процессе написания приложений для Windows, — это аутентификация и авторизация. *Аутентификация* подтверждает идентичность пользователя (но не самого пользователя как человека), а *авторизация* определяет круг задач, которые может выполнять пользователь (как внутри, так и вне приложения). Например, получение прав доступа к функциональной возможности приложения, которую этот пользователь не должен использовать, является нарушением безопасности внутри приложения. Удаление требуемого файла с использованием возможностей операционной системы является нарушением безопасности вне приложения.

Аутентификация с использованием входа в Windows

Самый лучший среди простых способ авторизации пользователя приложением — это использовать логин Windows. Существуют различные аргументы "за" и "против" для этой и других стратегий, но ключевым свойством является простота: простые вещи более безопасны.

Большая часть программного обеспечения, разработанного с помощью Visual Studio, будет использоваться в офисе пользователями, которые играют в компании различные роли. Например, некоторые пользователи могут работать в отделе продаж или бухгалтерии. Во многих средах наиболее привилегированными пользователями являются менеджеры или администраторы — вот еще один набор ролей. В большинстве офисов все сотрудники имеют собственные учетные записи и пользователям назначаются группы Windows, которые соответствуют тем ролям, которые они играют в компании.

ЗАПОМНИ!

Использование безопасности Windows работает только в том случае, если среда Windows настроена правильно. Вы не можете эффективно создать безопасное приложение в рабочем пространстве с множеством компьютеров с Windows XP, в котором все входят в систему с правами администратора, потому что вы просто не сможете сказать, кто из пользователей в какой роли находится.

Создать приложение для Windows так, чтобы воспользоваться преимуществами безопасности Windows, просто. Цель состоит в том, чтобы проверить, кто вошел в систему (аутентификация), а затем проверить роль этого пользователя (авторизация). Приведенные далее шаги показывают, как создать приложение, которое защищает систему меню для каждого пользователя, показывая и скрывая некоторые кнопки. Несмотря на то что рассматриваемый пример приложения основан на шаблоне Windows Forms App, описанные методы работают и с другими типами приложений, такими как приложение Windows Presentation Foundation (WPF). Чтобы успешно выполнить код, у вас должна быть среда, в которой есть группы пользователей Accounting, Sales и Management.

1. **Выберите пункт меню File⇨New⇨Project (Файл⇨Новый⇨Проект).**
   Вы увидите диалоговое окно нового проекта New Project.
2. **На левой панели выберите пункт Visual C#\Windows Classic Desktop.**
3. **На центральной панели выберите шаблон Windows Forms App.**
4. **Введите `SecureButton` в поле Name и щелкните на кнопке OK.**
   Visual Studio создаст новое приложение Windows Forms и откроет окно проектирования, в котором вы сможете добавить свои управляющие элементы.

5. **Добавьте в форму три кнопки — по одной для меню для каждой из групп: Sales Menu, Accounting Menu и Manager Menu.**

   На рис. 23.1 показан один из методов настройки формы. Изменить надпись на кнопке можно с помощью свойства Text в окне свойств Properties каждой кнопки.

Рис. 23.1. Приложение Windows Security

6. **Установите свойство (Name) для каждой кнопки так, чтобы оно соответствовало имени роли: SalesButton, AccountingButton и ManagerButton.**

   Давая кнопкам описательные имена, вы облегчаете работу с ними.

7. **Установите свойство видимости каждой кнопки равным False с тем, чтобы по умолчанию кнопки не были видимы на форме.**

8. **Дважды щелкните на форме для редактирования обработчика `Form1_Load`.**

9. **Перед инструкцией `namespace` импортируйте пространство имен `System.Security.Principal`:**

```
using System.Security.Principal;
```

10. **В обработчике `Form1_Load` инстанцируйте новый объект `Identity`, который представляет текущего пользователя, используя метод `GetCurrent` объекта `WindowsIdentity`, добавляя следующую строку кода:**

```
WindowsIdentity myIdentity = WindowsIdentity.GetCurrent();
```

11. **Получите ссылку на объект с помощью класса `WindowsPrincipal`:**

```
WindowsPrincipal myPrincipal = new WindowsPrincipal(myIdentity);
```

12. **Наведите указатель мыши на инструкции `using`.**

    При использовании C# 7.0 вы увидите пиктограмму лампочки. Эта пиктограмма говорит о том, что есть способы сделать ваш код более эффективным.

13. **Выберите пункт Remove Unnecessary Usings (удалить излишние директивы using).**

    Visual Studio удалит лишние инструкции `using`. Это позволит вашему коду быстрее загружаться и использовать ресурсы более эффективно.

14. **Также в подпрограмме `Form1_Load` добавьте небольшой код для определения, какая кнопка должна быть видимой.**

    Этот код приведен в листинге 23.1.

Листинг 23.1. Код приложения Windows Security

```
using System;
using System.Windows.Forms;
using System.Security.Principal;

namespace SecureButton
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            // Получение учетных данных пользователя
            WindowsIdentity myIdentity =
                    WindowsIdentity.GetCurrent();

            // Получение информации о правах пользователя
            WindowsPrincipal myPrincipal =
                new WindowsPrincipal(myIdentity);

            // Определение видимой кнопки
            // на основе прав пользователя
            if (myPrincipal.IsInRole("Accounting"))
            {
                AccountingButton.Visible = true;
            }
            else if (myPrincipal.IsInRole("Sales"))
            {
                SalesButton.Visible = true;
            }
            else if (myPrincipal.IsInRole("Management"))
            {
                ManagerButton.Visible = true;
            }
        }
    }
}
```

В некоторых случаях вам не нужна такая ролевая диверсификация. Иногда просто нужно знать, играет ли пользователь стандартную роль, которую обеспечивает `System.Security`. Используя перечислитель `WindowsBuiltInRole`, вы описываете действия, которые должны выполняться, когда, например, в систему входит администратор:

```
if (myPrincipal.IsInRole(WindowsBuiltInRole.Administrator))
{
    // Некоторые действия
}
```

Шифрование информации

Шифрование по своей сути является безумно сложным процессом. Имеется пять пространств имен, посвященных различным алгоритмам шифрования. Из-за сложности вопроса эта тема выходит за рамки книги, и мы не будем сколь-нибудь вдаваться в подробности. Тем не менее важно понимать, что один из ключевых элементов безопасности — шифрование файлов. Работая с файлом в приложении Windows Forms при отсутствии шифрования вы рискуете, что кто-нибудь загрузит его в текстовый редактор и просмотрит его содержимое.

В .NET в Visual Studio 2008 (C# 3.0) и более поздних реализован стандарт шифрования AES (Advanced Encryption Standard). Более старые версии Visual Studio полагаются на стандарт DES (Data Encryption Standard), который ныне не является самым надежным для 64-разрядных настольных машин. Используйте, где это возможно, AES, чтобы добиться наиболее высокого уровня надежности ваших приложений. Метод для шифрования с помощью DES находится в `DESCryptoServiceProvider` в пространстве имен `System.Security.Cryptography`.

Безопасность развертывания

Развертывая свое приложение с помощью ClickOnce, вам необходимо определить доступ к компьютеру, к которому будет обращаться приложение. ClickOnce — это стратегия развертывания на основе веб-сервера, которая позволяет пользователям запускать приложения Windows Forms из веб-браузера с помощью вкладки Windows Security в файле конфигурации, показанном на рис. 23.2. (Например, в рассмотренном примере проекта вы получаете доступ к этому диалоговому окну, выбирая пункт меню Project⇨SecureButton Properties.)

Здесь вы можете определить используемые вашим приложением функциональные возможности, чтобы устанавливающий его пользователь получал не ошибку системы безопасности при запуске приложения, а предупреждение при установке.

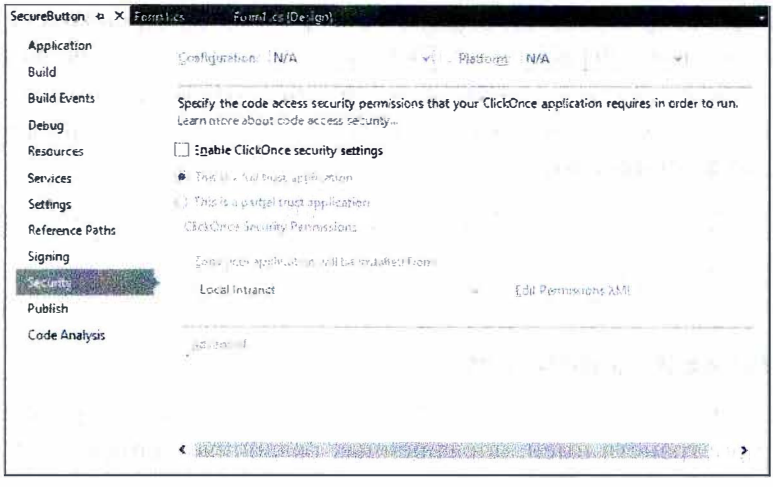


Рис. 23.2. Вкладка Windows Security файла конфигурации

Построение безопасных приложений Web Forms

Приложения Web Forms представляют собой отключенные от сети, слабо связанные программы, которые подвергают сервер риску потенциальных атак через открытые порты, используемые приложениями. *Слабо связанное* (loosely coupled) приложение связано с сервером отношением "транзакция-и-ожидание". Приложение отправляет запрос, а затем ожидает ответа.

Из-за этой связи обеспечение безопасности для приложений Web Forms становится более важным, чем когда-либо. Побочным эффектом является то, что ваше приложение может стать менее функциональным.

При создании веб-приложений вы тратите меньше времени на работу над аутентификацией (особенно если ваше приложение общедоступное) и больше — на защиту от хакеров. Поскольку вы делаете сервер общедоступным, ваши программы должны соответствовать совершенно новому набору правил безопасности.

Ключ к защите открытого сервера — честность. Вы должны быть честны с самим собой в отношении слабых сторон системы. Не пытайтесь успокаивать себя: "Хакер, конечно, может выяснить пароль с помощью XYZ, но никто никогда этого не сделает". Поверьте мне, кто-то это сделает. Вот два основных типа атак, о которых следует помнить при разработке приложения Web Forms:

- атаки SQL Injection;
- уязвимости сценариев.

Атаки SQL Injection

SQL Injection ("SQL-инъекция") происходит, когда в поле ввода, используемое для запроса базы данных в форме на веб-странице (такой, как текстовые поля имени пользователя и пароля в форме входа в систему) хакер вводит строку кода SQL. Вредоносный код SQL приводит к тому, что база данных работает некорректно или позволяет хакеру получить доступ, изменить или повредить базу данных.

Наилучший способ понять, как хакер использует SQL-инъекцию, — посмотреть пример. Предположим, веб-страница имеет код, который принимает от пользователя идентификатор товара в текстовом поле и возвращает сведения о товаре на основе этого идентификатора, введенного пользователем. Код на сервере может выглядеть следующим образом:

```
// Получение productId от пользователя
string ProductId = TextBox1.Text;

// Получение информации из базы данных
string SelectString = "SELECT * FROM Items WHERE ProductId = '" +
                      ProductId + "';";
SqlCommand cmd = new SqlCommand(SelectString, conn);
conn.Open();
SqlDataReader myReader = cmd.ExecuteReader();

// Обработка результата
myReader.Close();
conn.Close();
```

Обычно пользователь вводит в текстовое поле соответствующую информацию. Но хакер, в попытках применить инъекцию SQL, может ввести в `textBox1` следующую строку:

```
"FOOBAR';DELETE FROM Items;--"
```

Теперь код SQL, обрабатывающий данный запрос, будет выглядеть следующим образом:

```
SELECT * FROM Items WHERE ProductID = 'FOOBAR';DELETE FROM Items;--'
```

SQL Server неожиданно для вас выполняет некоторый код — в данном случае код, удаляющий все записи из таблицы Items.

ЗАПОМНИ!

Простейший способ предотвратить SQL-инъекцию — никогда не использовать конкатенацию строк при генерации SQL-запроса. Используйте хранимые процедуры и параметры SQL. Вы можете прочитать больше об этом в главе 24, "Обращение к данным".

Уязвимости сценариев

Использование *уязвимости сценария* (script exploit) — это уязвимость в системе безопасности, которая использует механизм JavaScript в веб-браузере пользователя. Уязвимость сценариев использует одну из наиболее распространенных функций общедоступных приложений Web Forms — обеспечение взаимодействия между пользователями. Например, приложение Web Forms может позволить пользователю опубликовать комментарий, который могут просматривать другие пользователи сайта, или может позволить пользователю заполнить онлайн-профиль.

Вставив в профиль или комментарий некоторый код сценария, злонамеренный пользователь может захватить управление браузером следующего пользователя, который зайдет на сайт. Возможны несколько вариантов результата такого перехвата браузера, и ни один из них не назовешь хорошим.

Например, когда пользователь заходит на ваш сайт, для JavaScript доступна коллекция файлов cookie. Злоумышленник может вставить в профиль некоторый код сценария, копирующий файл cookie для вашего сайта на свой удаленный сервер. Это может дать злоумышленнику доступ к текущему сеансу пользователя, так как идентификатор сеанса хранится в виде cookie, и тем самым злоумышленник может подделать учетные данные текущего пользователя.

К счастью, ASP.NET не позволяет пользователям вводить большие фрагменты кода сценария в поле формы и отправлять его на сервер. Попробуйте сделать это с помощью базового проекта Web Forms, выполнив следующие действия (вы увидите ошибку, показанную ниже, на рис. 23.4).

1. **Выберите пункт меню File⇨New⇨Project (Файл⇨Новый⇨Проект).**

   Вы увидите диалоговое окно нового проекта New Project.

2. **На левой панели выберите Visual C#\Web.**

3. **Выберите ASP.NET Web Application (веб-приложение ASP.NET) на центральной панели.**

4. **Введите `SecureForm` в поле Name и щелкните на кнопке OK.**

   Visual Studio выведет диалоговое окно New ASP.NET Web Application наподобие показанного на рис. 23.3.

5. **Выберите Web Forms и щелкните на кнопке OK.**

   Visual Studio создаст новое приложение Web Forms.

6. **Дважды щелкните на `Default.aspx` в Solution Explorer.**

   Вы увидите окно дизайнера форм. Дизайнер форм содержит все виды элементов управления, но пока что не беспокойтесь о них.

7. **Выберите вкладку Design и добавьте текстовое поле и кнопку на страницу по умолчанию.**

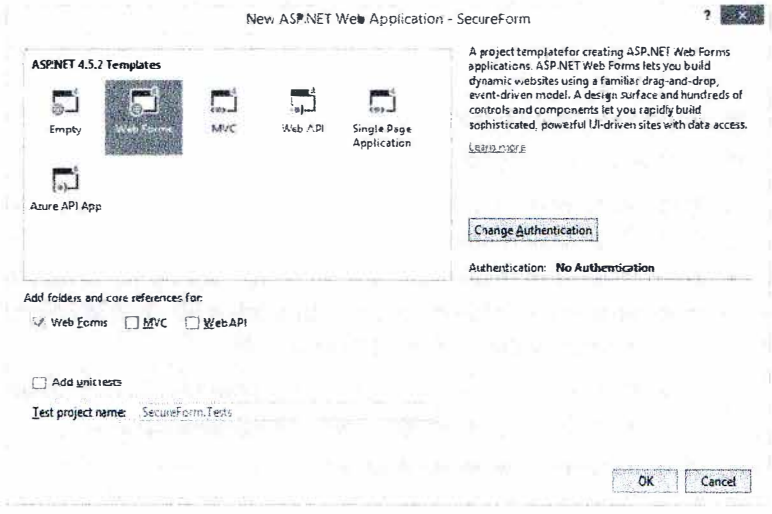


Рис. 23.3. Выберите тип приложения из списка

8. **Запустите проект.**
9. **Введите в текстовое поле `<script>msgbox()</script>`.**
10. **Щелкните на кнопке.**

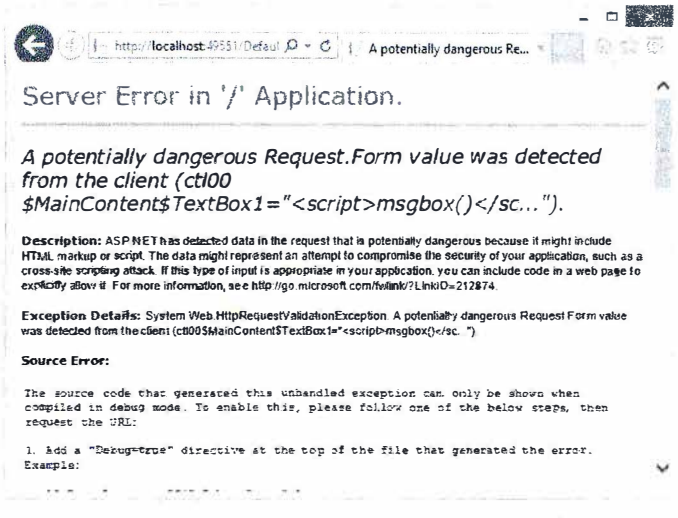


Рис. 23.4. По умолчанию уязвимости скриптов блокированы

Наилучшие методы защиты приложений Web Forms

Помимо гарантии того, что ваше приложение Web Forms будет предотвращать атаки SQL-инъекций и использование уязвимости сценариев, следует помнить о некоторых полезных методах защиты ваших веб-приложений.

В следующем списке приведены некоторые из наиболее важных методов их защиты.

- Регулярно обновляйте Internet Information Server (IIS).
- Выполняйте резервное копирование всего, что можно.
- Избегайте использования переменной `Querystring` (значения после имени страницы в URL).
- Не оставляйте комментарии в HTML. Любой пользователь может просмотреть HTML-код и просмотреть ваши комментарии, выбрав просмотр исходного текста в браузере.
- При обеспечении безопасности не полагайтесь на проверку на стороне клиента — она может быть фальсифицирована.
- Используйте надежные пароли.
- Не полагайтесь на то, что пользователь отправил вам информацию из вашей формы и она безопасна. Форму легко подделать.
- Убедитесь, что сообщения об ошибках не дают пользователю никакой информации о вашем приложении. Отправляйте сообщения об ошибках по электронной почте, а не отображайте их пользователю.
- Используйте Secure Sockets Layer (SSL).
- Не храните ничего важного в виде cookie.
- Закройте все неиспользуемые порты вашего веб-сервера.
- Отключите SMTP в IIS, если только вы в нем не нуждаетесь.
- Если вы разрешаете загрузку — выполняйте проверку на вирусы.
- Не запускайте ваши приложения с правами администратора.
- По возможности используйте временные cookie, устанавливая дату истечения их срока годности. Оставляйте файлы cookie активными только на время сеанса.
- Ограничьте размер загрузок файлов. Это можно сделать в файле конфигурации `Web.Config`:

```
<configuration>
    <system.web>
        <httpRuntime maxRequestLength="4096"/>
    </system.web>
</configuration>
```

- Помните, что `ViewState` в Web Forms легко обнаруживается.

Использование System.Security

Хотя многие инструменты безопасности встроены в классы, которые их используют, некоторые классы не поддаются описанию или классификации. По этой причине System.Security представляет собой хранилище тех инструментов, которые не получается классифицировать более точно.

Наиболее распространенные пространства имен System.Security описаны в табл. 23.1. Использование пространства имен Security.Principal было продемонстрировано ранее в этой главе.

Таблица 23.1. Распространенные пространства имен в System.Security

Пространство имен	Описание	Основные классы
Security	Базовые классы безопасности	CodeAccessPermission, SecureString
AccessControl	Интеллектуальный контроль авторизации	AccessRule, AuditRule
Authorization	Перечисления, описывающие безопасность приложения	CipherAlgorithmType
Cryptography	Содержит несколько пространств имен, которые помогают с шифрованием	CryptoConfig, DESCryptoServiceProvider
Permissions	Управляет доступом к ресурсам	PrincipalPermission, SecurityPermission
Policy	Создание правил системы стратегий безопасности среды выполнения	Evidence, Site, Url
Principal	Определяет объект, представляющий текущий пользовательский контекст	WindowsIdentity, WindowsPrincipal

Глава 24

Обращение к данным

В ЭТОЙ ГЛАВЕ...

- Пространство имен `System.Data`
- Подключение к источникам данных
- Изучение Entity Framework
- Работа с данными из баз данных

Вероятно, вы обнаружите, что доступ к данным является наиболее важной частью вашего использования .NET Framework. И скорее всего, вы будете использовать различные функции пространства имен `System.Data` чаще, чем из любого другого пространства имен.

Несомненно, одним из наиболее распространенных применений Visual Studio является создание бизнес-приложений. Бизнес-приложения — это приложения для работы с данными. Желательно понемногу знать все стороны программирования на C#, но при создании бизнес-приложений необходимо иметь полное понимание пространства имен `System.Data`.

До тех пор, пока в 2003 году платформа .NET Framework не стала популярной, большинство бизнес-приложений, созданных с использованием продуктов Microsoft, использовали FoxPro или Visual Basic. За последние несколько лет C#, несомненно, заменил эти языки в качестве основного языка бизнес-программиста. Вы можете рассматривать инструменты для работы с данными в C# с трех сторон.

» **Подключение к базе данных.** Получение информации из базы данных и запись информации в нее является основным содержимым пространства имен `System.Data`.

» **Хранение данных в контейнерах в ваших программах.** Контейнеры `DataSet`, `DataView` и `DataTable` представляют собой полезные механизмы для хранения данных. Если вы программист на Visual Basic 6 или ASP, то можете вспомнить `Recordset`, замененный новыми конструкциями.


ЗАПОМНИ!

Язык интегрированных запросов (Language Integrated Query — LINQ) позволяет получать данные из контейнеров данных с использованием языка структурированных запросов (Structured Query Language — SQL), а не сложного объектно-ориентированного языка.

» **Интеграция с управляющими элементами данных.** Пространства имен `System.Web` и `System.Windows` обеспечивают интеграцию с элементами управления данными. Интеграция управления данными интенсивно использует подключение к базе данных и контейнеры данных. Это делает управляющие элементы данных одной из главных тем данной главы.

Знакомство с `System.Data`

Данные в .NET отличаются от данных в любой другой платформе Microsoft. Microsoft продолжает изменять способ работы с данными в .NET Framework. ADO.NET (реализация которого содержится в новой библиотеке `System.Data`) и предоставляет еще один новый способ рассмотрения данных с точки зрения разработки.

» **Отключение**. После получения информации из источника данных ваша программа больше не подключается к этому источнику данных. У вас есть копия данных. Это решает одну проблему и тут же вызывает другую.

- У вас больше нет проблемы блокировки строк. Поскольку вы работаете с копией данных, вам не нужно ограничивать внесение изменений в базу данных.
- Вы получаете проблему *последнего победителя*. Если два экземпляра программы получают одинаковые данные и оба их обновляют, то тот экземпляр, который вносит измененные данные в базу данных последним, перезаписывает изменения, внесенные первым экземпляром.

- **Управление XML.** Копия данных, полученная из источника, представляет собой текст XML. Он может быть конвертирован в произвольный формат, когда Microsoft сочтет это необходимым для повышения производительности, но в любом случае это всего лишь XML, что значительно упрощает перенос между платформами, приложениями или базами данных.
- **Обобщенные контейнеры баз данных.** Контейнеры никак не зависят от типа базы данных — их можно использовать для хранения данных из любого источника.
- **Адаптеры, специфичные для базы данных.** Подключения к базе данных зависят от ее платформы, поэтому, если вы хотите подключиться к конкретной базе данных, вам нужны компоненты, которые работают с этой конкретной базой данных.

Процесс получения данных тоже немного изменился. Раньше у вас были соединение и команда, которая возвращала набор записей. Теперь у вас есть адаптер, который использует соединение и команду для заполнения контейнера `DataSet`. Изменился способ, которым пользовательский интерфейс помогает вам выполнить вашу работу.

`System.Data` имеет классы, которые помогут вам подключиться к множеству различных баз данных и других типов данных. Эти классы разделены по пространствам имен в табл. 24.1.

Таблица 24.1. Пространства имен `System.Data`

Пространство имен	Назначение	Наиболее часто используемые классы
`System.Data`	Общие классы ADO.NET	Контейнеры `DataSet`, `DataView`, `DataTable`, `DataRow`
`System.Data.Common`	Служебные классы, используемые классами конкретных баз данных	`DbCommand`, `DbConnection`
`System.Data.ODBC`	Классы для подключения к базам данных ODBC, таким как dBASE	`OdbcCommand`, `OdbcAdapter`
`System.Data.OleDb`	Классы для подключения к базам данных OleDb, таким как Access	`OleDbCommand`, `OleDbAdapter`

Окончание табл. 24.1

Пространство имен	Назначение	Наиболее часто используемые классы
`System.Data.OracleClient`	Классы для подключения к базам данных Oracle	`OracleCommand`, `OracleAdapter`
`System.Data.SqlClient`	Классы для подключения к Microsoft SQL Server	`SqlCommand`, `SqlDataAdapter`
`System.Data.SqlTypes`	Классы для обращения к типам SQL Server	`SqlDateTime`

Хотя в пространстве имен `System.Data` имеется много других функциональных возможностей, в этой главе основное внимание уделяется тому, как Visual Studio реализует эти инструменты. В предыдущих версиях программного обеспечения для разработки визуальные инструменты только усложняли ситуацию из-за проблемы черного ящика.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Проблема черного ящика заключается в том, что среда разработки делает некоторые вещи, которые вы не можете контролировать. Иногда, когда что-то делается вместо вас, это хорошо; но когда среда разработки создает нечто не в точности так, как это нужно вам, сгенерированный код оказывается бесполезным.

К счастью, в настоящее время это уже не так. Теперь при использовании инструментов визуальных данных Visual Studio генерирует полностью открытый и понятный код C#. Вы должны быть довольны результатами.

Классы данных и каркас

Классы данных предназначены для хранения информации. В части 1, "Основы программирования на C#", рассказывается о коллекциях, которые предназначены для хранения информации во время работы приложения. Еще одним примером хранения информации являются хеш-таблицы. *Коллекции* содержат списки объектов, а *хеш-таблицы* — пары имен и значений. Контейнеры данных содержат большие количества информации и помогают манипулировать ею. Вот список контейнеров данных.

- `DataSet`. Будучи чем-то наподобие дедушки всех остальных контейнеров, `DataSet` является представлением всей базы данных в памяти.
- `DataTable`. Это единственная таблица данных, хранящаяся в памяти, и наиболее близкая к `Recordset` (если вы программист на VB 6 и ищете что-то похожее). Контейнеры `DataSet` состоят из контейнеров `DataTable`.
- `DataRow`. Очевидно, что это — строка в контейнере `DataTable`.
- `DataView`. Копия `DataTable`, которую можно использовать для сортировки и фильтрации данных для просмотра.
- `DataReader`. Доступный только для чтения однонаправленный поток данных, используемый для одноразовых процессов, таких как заполнение списков. Обычно называется *пожарным шлангом* (*fire hose*).

Получение данных

В пространстве имен `System.Data` все вращается вокруг получения данных из базы данных, такой как Microsoft SQL Server, и заполнения контейнеров данных. Вы можете получить эти данные вручную. Вообще говоря, процесс идет примерно следующим образом.

1. Вы создаете адаптер.
2. Вы говорите адаптеру, как получить информацию из базы данных (подключение).
3. Адаптер подключается к базе данных.
4. Вы говорите адаптеру, какую информацию нужно получить из базы данных (команда).
5. Адаптер заполняет контейнер `DataSet` данными.
6. Закрывается подключение между адаптером и базой данных.
7. Теперь в вашей программе имеется отсоединенная копия данных.

Вообще говоря, вам вовсе не обязательно проходить весь этот процесс — если вы позволите, Visual Studio многое сделает вместо вас. Лучшая практика — максимально использовать автоматизацию.

Использование пространства имен System.Data

Пространство имен System.Data — это еще одно пространство имен между миром кода и миром визуальных инструментов, благодаря которому взаимосвязь между элементами управления формы и пространством имен Data выглядит так, как будто данные находятся прямо внутри элементов управления, в особенности при работе с Windows Forms.

В следующих разделах главным образом рассматриваются визуальные инструменты, которые являются такой же частью работы с C#, как и код. Сначала вы узнаете, как подключиться к источникам данных, а затем увидите, как написать быстрое приложение, используя одно из этих подключений.

Чтобы заставить все это работать, нужна определенная схема вашей базы данных, например созданный собственноручно локальный проект или готовый образец схемы.

Настройка образца схемы базы данных

Для начала обратитесь по адресу http://msftdbprodsamples.codeplex.com/releases/view/55330. Если этот URL не работает, выполните поиск *SQL Server 2012 samples* и найдите ближайшую ссылку CodePlex.

ИСПОЛЬЗОВАНИЕ БАЗ ДАННЫХ СТАРЫХ ВЕРСИЙ

Вы можете удивиться, почему в книге не используется последняя версия SQL Server. Дело в том, что использование более старой базы данных AdventureWorks 2012 упрощает нашу задачу, поскольку к ней можно без проблем обращаться непосредственно из C#. Использовать при обучении как можно более простые решения — это всегда хорошая идея, и именно она принята как руководящая в данной книге.

На этой странице имеется целый ряд примеров листингов, примеров приложений и схем, отчетов баз данных, фрагментов онлайновой обработки транзакций (Online Transaction Processing — OLTP) и множество других вещей. Примеры приложений представляют собой полнофункциональные приложения, которые демонстрируют полную реализацию с использованием .NET программного обеспечения, управляемого данными. Одни из этих приложений разработаны на C#, другие — на Visual Basic. Примеры схем представляют собой только базы данных и предназначены для администраторов баз данных, позволяя им на практике получить опыт работы с системой.

Все примеры схем вполне работоспособны. Если вы хотите использовать точно ту же схему, что и в приведенных здесь примерах, выберите AdventureWorks2012 Data File. Но, возможно, для вашей работы лучше подойдут другие варианты. Для инсталляции схемы загрузите файл MDF и поместите его туда, где вам будет удобно. В конечном итоге вы будете ссылаться на него в своем проекте, так что такое локальное расположение, как `C:\Databases`, может оказаться хорошим выбором. Если вы знакомы с SQL Server, то можете добавить базу данных к вашей локальной установке и указать на нее. Если вы не являетесь администратором базы данных, то можете указать провайдеру данных файл непосредственно. Именно такой подход используется в оставшейся части этой главы.

Подключение к источнику данных

В наши дни подключение к базе данных — это не просто установление подключения к SQL Server. Разработчики C# должны подключаться к мэйнфреймам, текстовым файлам, необычным базам данных, веб-сервисам и другим программам. Все эти разрозненные системы интегрируются в окна и веб-экраны с функциями создания, чтения, обновления и удаления (create, read, update and delete — CRUD).


ЗАПОМНИ!

Доступ к этим источникам данных в основном зависит от классов `Adapter` индивидуальных пространств имен базы данных. Oracle имеет собственное пространство имен, так же как и SQL Server. Базы данных, совместимые с ODBC (Open Database Connectivity) (например, Microsoft Access), имеют собственные классы адаптеров; более новый протокол OLEDB (Object Linking and Embedded Database) также имеет свои классы.

К счастью, большую часть проблем решает Мастер настройки источника данных (Data Source Configuration Wizard), доступный на панели Data Sources, на которой, работая с данными, вы проводите бо́льшую часть времени. Чтобы начать работу с мастером настройки источника данных, выполните следующие действия.

1. **Выберите пункт меню создания нового проекта File⇨New⇨Project.**
   При этом откроется диалоговое окно нового проекта New Project.
2. **На левой панели выберите Visual C#\Windows Classic Desktop.**
3. **Выберите Windows Forms App на центральной панели.**
4. **Введите `AccessData` в поле Name и щелкните на кнопке OK.**

Visual Studio создаст новое приложение Windows Forms (известное также как WinForms) и выведен на экран дизайнер форм, в котором вы сможете добавлять в форму управляющие элементы. На этом этапе вы можете создать источник данных для использования в этом примере приложения.

5. **Выберите пункт меню View⇨Other Windows⇨Data Sources (Вид⇨Прочие окна⇨Источники данных) или нажмите клавиши <Shift+Alt+D>.**

   Панель источников данных Data Sources сообщит вам об отсутствии у вас источников данных.

6. **Щелкните на ссылке добавления нового источника данных Add New Data Source на панели Data Sources.**

   Вы увидите мастер Data Source Configuration Wizard, показанный на рис. 24.1. Мастер предлагает выбрать тип источника данных. Наиболее интересным из них является источник Object, который предоставляет вам доступ к объекту в сборке для привязки ваших элементов управления.

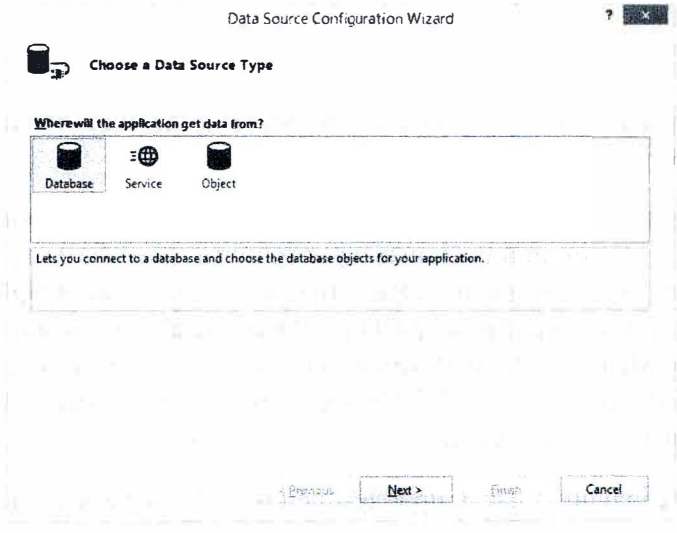


Рис. 24.1. Выберите тип источника данных приложения

7. **Выберите тип источника данных Database и щелкните на кнопке Next.**

   Вы увидите диалоговое окно выбора модели базы данных, показанное на рис. 24.2. Количество возможных вариантов выбора зависит от используемой версии Visual Studio. Как минимум вы получите доступ к модели Dataset.

8. **Выберите модель Dataset и щелкните на кнопке Next.**

   Вы увидите диалоговое окно выбора подключения данных, показанное на рис. 24.3. Поскольку это новое приложение, вы не должны увидеть ни одного подключения.

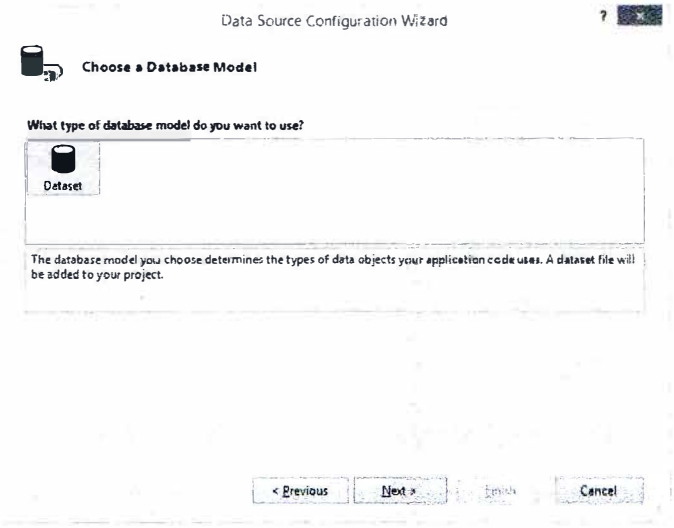


Рис. 24.2. Выберите модель базы данных

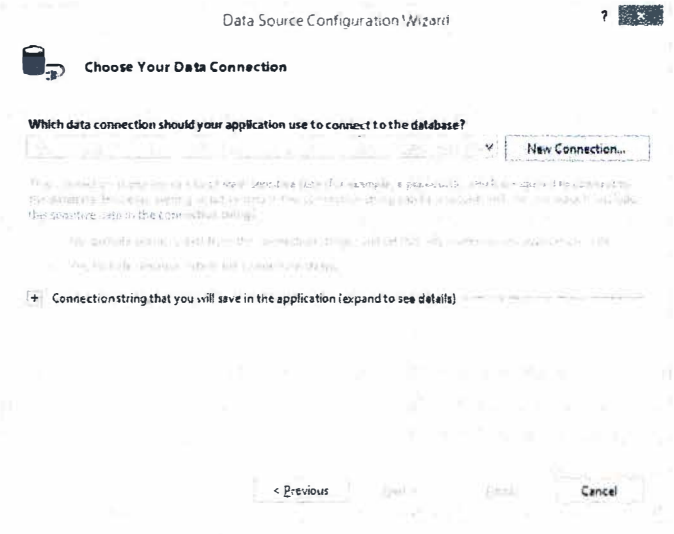


Рис. 24.3. Выбор подключения к данным

9. **Щелкните на кнопке нового подключения New Connection.**

Visual Studio предложит вам создать новое подключение с использованием диалогового окна выбора источника данных Choose Data Source, показанного на рис. 24.4. Наш пример основан на прямом подключении к Microsoft SQL Server Database File, который представляет собой простейший способ создания подключения. Обратите внимание, что можно также создать непосредственное подключение к файлам базы данных Microsoft Access, а также подключения к другим базам данных с помощью соответствующего адаптера базы данных.

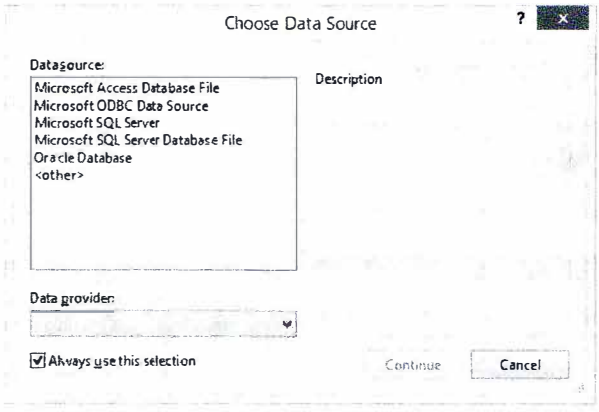


Рис. 24.4. Диалоговое окно Choose Data Source

СОВЕТ

Поле провайдера данных Data Provider может содержать несколько провайдеров данных. Мастер обычно выбирает наиболее эффективного провайдера данных. Однако другие провайдеры данных могут иметь функции, которые потребуются вам для определенного типа приложения. Всегда убеждайтесь, что вы выбираете наилучшего провайдера данных для потребностей вашего конкретного приложения.

Следующие шаги относятся к использованию файла базы данных Microsoft SQL Server. Другие типы источников данных могут потребовать выполнения других шагов для создания соединения.

10. **Выберите Microsoft SQL Server Database File и щелкните на кнопке Continue.**

    Вы увидите диалоговое окно добавления подключения Add Connection, показанное на рис. 24.5.

11. **Щелкните на кнопке Browse для вывода диалогового окна выбора файла базы данных Select SQL Server Database File, выберите в нем загруженный ранее файл `AdventureWorks2012_Data.mdf` и щелкните на кнопке Open.**

    Мастер добавит выбранный вами файл в поле Database File Name.

12. **Щелкните на кнопке OK.**

    Visual Studio может предложить вам обновить файл базы данных, что вполне нормально. Просто щелкните на кнопке Yes для завершения процесса. Через несколько секунд вы увидите подключение, добавленное в диалоговое окно Data Source Configuration Wizard, показанное ранее, на рис. 24.5.

13. **Щелкните на кнопке Next.**

    Мастер может спросить, хотите ли вы скопировать файл данных в свой текущий проект. Если вы работаете с этой книгой в изолированном проекте, это нормально. Если вы занимаетесь разработкой в команде, убедитесь, что это соответствует методологии жизненного цикла. В данном примере

щелкните на кнопке No, потому что вы единственный, кто использует этот источник данных, и нет веской причины для создания его копии. Мастер отображает имя файла строки подключения, как показано на рис. 24.6, и спрашивает, хотите ли вы сохранить его в приложении.

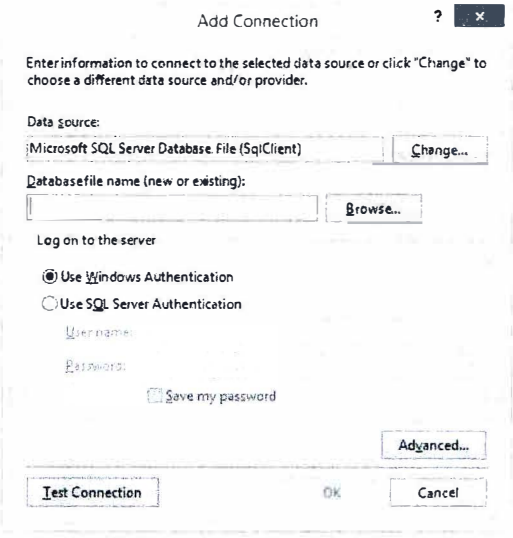


Рис. 24.5. Укажите местоположение файла базы данных

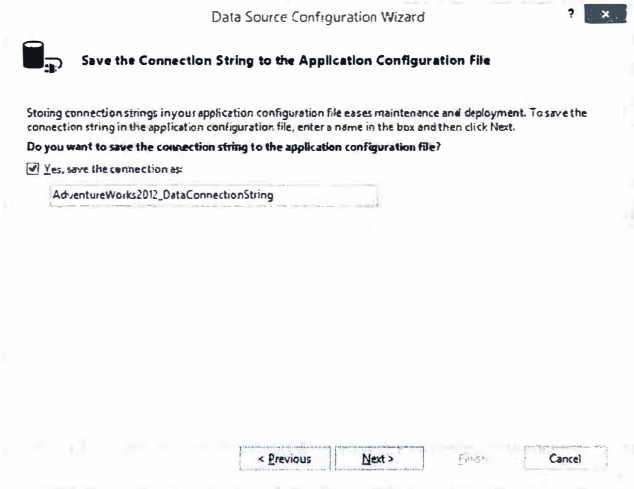


Рис. 24.6. Сохраните файл со строкой подключения для использования базы данных в своем приложении

14. **Щелкните на кнопке Next.**

Вы увидите диалоговое окно выбора объектов базы данных. Вы можете выбрать в нем таблицы, представления или хранимые процедуры, которые планируете использовать.

15. **В списке таблиц выберите Product и ProductCategory.**

Диалоговое окно выбора объектов базы данных должно иметь такой вид, как показано на рис. 24.7.

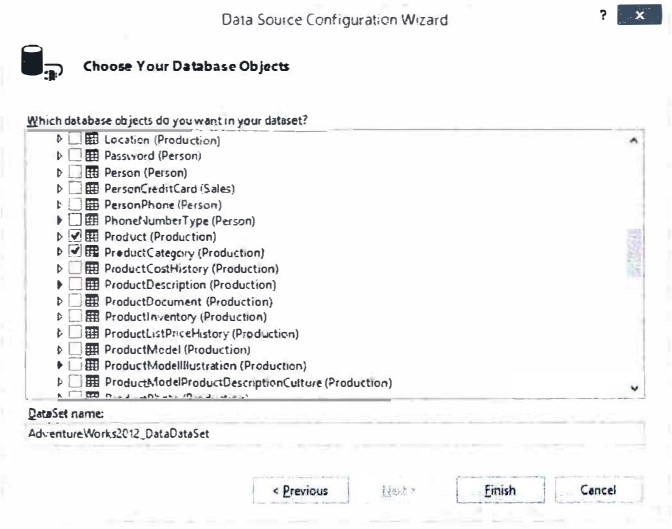


Рис. 24.7. Выбор объектов данных

16. **Щелкните на кнопке Finish.**

Работа завершена. Если вы взглянете на панель Data Sources, то увидите, что в ваш проект добавлен `DataSet` с двумя запрошенными вами таблицами, как показано на рис. 24.8.

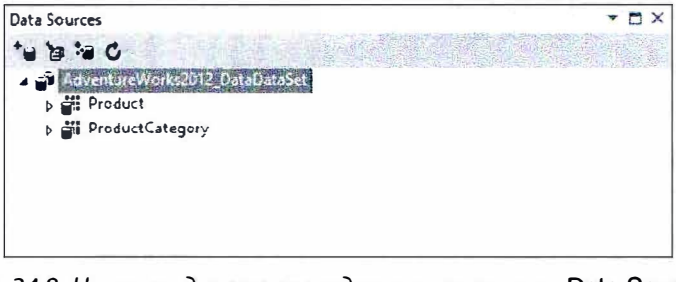


Рис. 24.8. Новые подключения к данным на панели Data Sources

Выполнив предыдущие шаги, вы создадите в Visual Studio две важные сущности.

- Соединение с базой данных, показанное в Server Explorer.
- Набор данных, специфичный для данного проекта, которого не будет в обозревателе при начале другого проекта.

Обе они важны, предоставляя различную функциональность. В этой главе мы сосредоточимся на конкретном источнике данных проекта, выводимом с помощью набора данных.

Работа с визуальными инструментами

Инструменты быстрой разработки (Rapid Application Development — RAD) для данных в C# в Visual Studio создают для вас необходимые заготовки кода. Выберите панель Data Sources (пункт меню View⇨Other Windows⇨Data Sources) и щелкните на таблице на панели. Вы увидите выпадающий список со стрелкой справа, показанный на рис. 24.9. Щелкните на стрелке, и вы увидите раскрывающийся список, в котором сможете выбрать способ интеграции этой таблицы в Windows Forms.

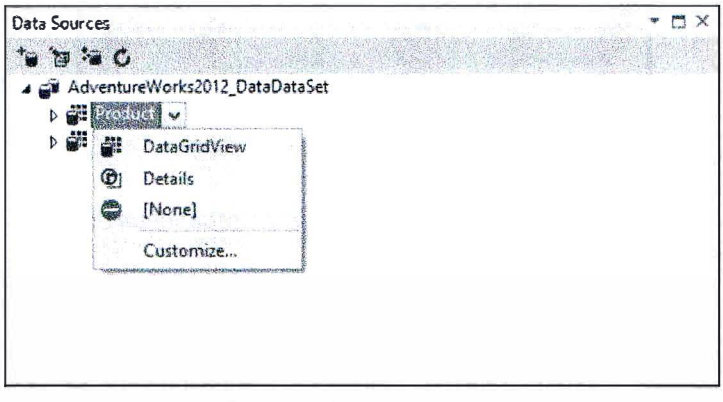


Рис. 24.9. Выпадающий список настройки таблицы

Измените Product table на Details View. Так вы сможете создать детальное представление, которое позволяет пользователям легко просматривать и изменять данные. Затем перетащите таблицу в форму, и для вас будет создано представление Details View, как показано на рис. 24.10 (вся форма не показана, потому что она слишком длинная).

Когда вы отпускаете таблицу в форме, выполняется ряд действий.

- Добавляются поля и их имена.
- Поля имеют наиболее подходящий формат.
- Имя поля представляет собой метку.
- Visual Studio автоматически добавляет пробел там, где изменяется регистр символов.

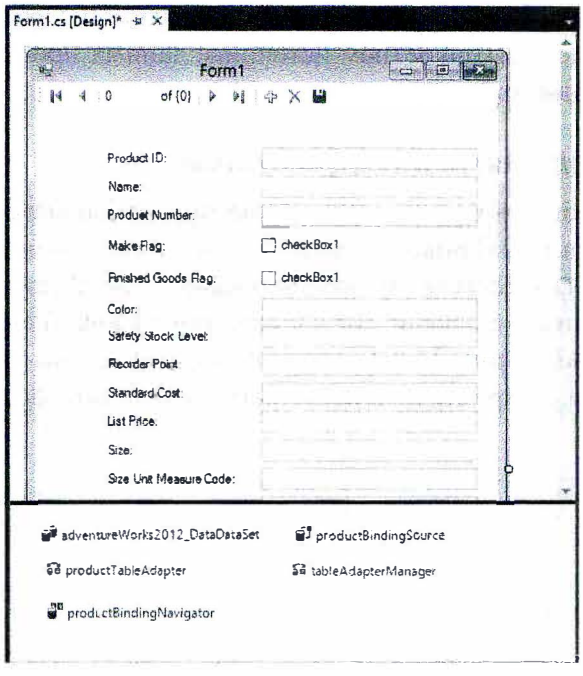


Рис. 24.10. Создание детального представления данных

СОВЕТ

Обратите внимание, что каждое поле получает дескриптор `SmartTag`, который позволяет указать запрос для значений в текстовом поле. Вы также можете предварительно настроить используемый управляющий элемент, изменив значения на панели источников данных Data Sources (см. рис. 24.9). В область уведомлений Component Tray внизу страницы добавляются пять полностью основанных на коде объектов: `DataSet` (`adventureWorks2012_DataDataSet`), `BindingSource` (`productBindingSource`), `TableAdapter` (`productTableAdapter`), `TableAdapterManager` (`tableAdapterManager`) и `BindingNavigator` (`productBindingNavigator`).

В верхнюю часть страницы добавляется панель VCR Bar (технически именуемая `BindingNavigator`). При запуске приложения вы можете использовать ее для циклического перехода между записями таблицы. Щелкните на кнопке Start, чтобы увидеть работу панели. Вы можете без проблем просматривать элементы в базе данных, как показано на рис. 24.11.

Написание кода для работы с данными

Однако в большинстве сред разработки уровня предприятия вы не будете использовать визуальные инструменты для создания программного обеспечения

доступа к данным. Поскольку корпоративное программное обеспечение часто предъявляет особые требования, как правило, соответствующая инфраструктура уже имеется, и самый простой способ управлять этими спецификациями — использовать уникальный и настраиваемый код. Короче говоря, некоторые организации не хотят, чтобы все было так, как делает Microsoft.

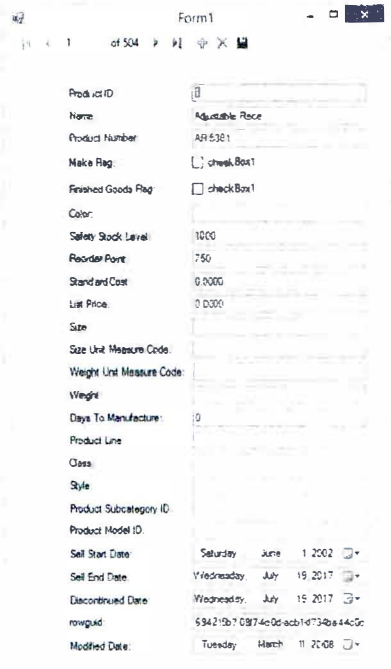


Рис. 24.11. Работающее приложение

Вывод визуальных инструментов

Зачастую в корпоративных средах визуальные инструменты не используются, потому что код, который они создают, довольно сложен. Щелкните на `Form1.Designer.cs` в обозревателе решений дважды, чтобы увидеть код для управляющих элементов формы. На рис. 24.12 показано, что вы видите, когда впервые попадаете сюда. Поле в верхней части окна кода помечает свернутую область кода Windows Form Designer generated code, и вы не можете не обратить внимание, что она содержит свыше семи сотен строк. Это очень немало. В этом коде нет ничего неверного; просто он специально сделан максимально обобщенным, чтобы обеспечить поддержку всего, что кто-то может захотеть с ним сделать. Корпоративные клиенты часто хотят гарантировать, что все сделано единообразно, и по этой причине часто определяют конкретный формат кода данных и ожидают, что разработчики программного обеспечения будут использовать именно его, а не визуальные инструменты.

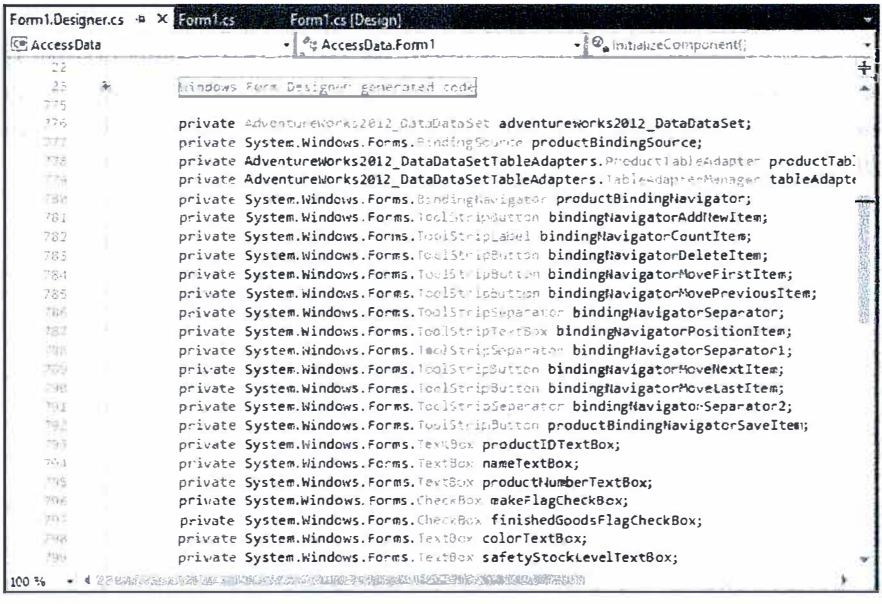


Рис. 24.12. Сгенерированный код. Ваши впечатления?

Базовый код данных

Код нашего примера проекта прост.

```
using System;
using System.Windows.Forms;
namespace AccessData
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void productBindingNavigatorSaveItem_Click(
                        object sender, EventArgs e)
        {
            this.Validate();
            this.productBindingSource.EndEdit();
            this.tableAdapterManager.UpdateAll(
                    this.adventureWorks2012_DataDataSet);
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // TODO: Эта строка кода загружает данные в таблицу
            // 'adventureWorks2012_DataDataSet.Product'.
            // При необходимости вы можете переместить
```

```
            // или удалить его.
            this.productTableAdapter.Fill(
                this.adventureWorks2012_DataDataSet.Product);
        }
    }
}
```

Хотя этот код довольно прост, очевидно, что это еще не все, что вам нужно. Остальная часть кода находится в файле, который генерирует саму визуальную форму, поддерживая визуальные компоненты.

Может наступить момент, когда вы захотите подключиться к базе данных без использования визуальных инструментов. Ранее в этой главе уже рассматривались необходимые для этого шаги, и вот как выглядит код, который при этом получается.

```
1. SqlConnection mainConnection = new SqlConnection();
2. mainConnection.ConnectionString = "server=(local);"+
                                     "database=Assets_Maintenance;"+
                                     "Trusted_Connection=True"
3. SqlDataAdapter partsAdapter = new SqlDataAdapter(
                              "SELECT * FROM Parts", mainConnection)
4. DataSet partsDataSet = new DataSet();
5. mainConnection.Open();
6. partsAdapter.Fill(partsDataSet);
7. mainConnection.Close();
```

СОВЕТ

Этот подход особенно полезен, когда вы хотите создать веб-сервис или библиотеку классов, хотя следует помнить, что в этих типах проектов все еще можно использовать визуальные инструменты. В следующих абзацах строка за строкой обсуждается показанный выше код.

Строка 1 устанавливает новое соединение для передачи данных, а строка 2 заполняет его строкой соединения. Вы можете получить ее у администратора базы данных (DBA) или на панели свойств подключения к данным.

Строка 3 содержит SQL-запрос. В главе 23, "Написание безопасного кода", говорилось о том, что такая методика является не лучшим выбором с точки зрения безопасности и что следует использовать хранимые процедуры. Хранимая процедура представляет собой артефакт базы данных, который позволяет использовать не динамически генерируемые строки SQL, а параметризованный запрос ADO.NET. Никогда не используйте встроенный SQL в производственных системах.

Строка 4 создает новый набор данных. Здесь хранится схема возвращаемых данных; этот набор данных вы будете использовать для навигации по данным.

Строки 5–7 выполняют всю основную работу: открывают соединение, связываются с базой данных, заполняют набор данных с помощью адаптера и

затем закрывают базу данных. В этом простом примере все просто, но более сложные примеры создают более сложный код.

После выполнения этого кода в контейнере `DataSet` у вас будет таблица `Products`, так же как и при использовании визуальных инструментов. Чтобы получить доступ к информации, вы должны установить значение текстового поля равным значению ячейки в контейнере DataSet, например:

```
TextBox1.Text = myDataSet.Tables[0].Rows[0]["name"]
```

Чтобы перейти к следующей записи, вам нужно написать код, который изменит `Rows[0]` на `Rows[1]`. Как видите, получается довольно объемный код. Вот почему мало кто использует базовый код данных для получения информации из баз данных. Либо вы используете визуальные инструменты, либо вы используете некоторую объектно-реляционную модель, такую как Entity Framework.

Использование Entity Framework

Объектные модели (которые рассматриваются в большей части этой книги) и базы данных не сочетаются между собой. Это два разных подхода к одной и той же информации. В основном проблема заключается в наследовании, которое обсуждается в части 2, "Объектно-ориентированное программирование на C#". Если у вас есть класс с именем `ScheduledEvent`, который имеет определенные свойства, и набор классов, наследующих его, таких как `Courses`, `Conferences` и `Parties`, то просто не существует хорошего способа показать это отношение в базе данных реляционного типа.

Если вы создадите большую таблицу для `ScheduledEvents` со всеми возможными типами свойств и просто добавите свойство `Type`, чтобы можно было отличать `Courses` от `Parties`, в таблице будет много пустых ячеек. Если вы создаете таблицу только для свойств, находящихся в `ScheduledEvents`, а затем отдельно создаете таблицы для `Courses` и `Parties`, то вы делаете базу данных поразительно сложной. Чтобы решить эту проблему, Microsoft создала Entity Framework. Глобально цель заключается в том, чтобы сначала спроектировать базу данных, а затем автоматически создать объектную модель для работы с ней и поддерживать ее актуальность при изменениях таблиц.

Объектно-ориентированная технология Entity Framework выполняет свою часть работы в этом процессе, генерируя контекст, который можно использовать для связи с данными способом, который больше похож на объектную модель, чем на базу данных.

Генерация объектной модели

Для начала вам нужна сама модель. Просто выполните следующие шаги, чтобы сгенерировать объектную модель.

1. **Создайте новый проект.**

   Я использовал проект Windows Forms под названием "EntityFramework".

2. **Щелкните правой кнопкой мыши на записи EntityFramework в обозревателе решений Solution Explorer и выберите Add⇨New Item в контекстном меню. Выберите на левой панели папку Data.**

   Вы увидите диалоговое окно Add New Item, показанное на рис. 24.13.

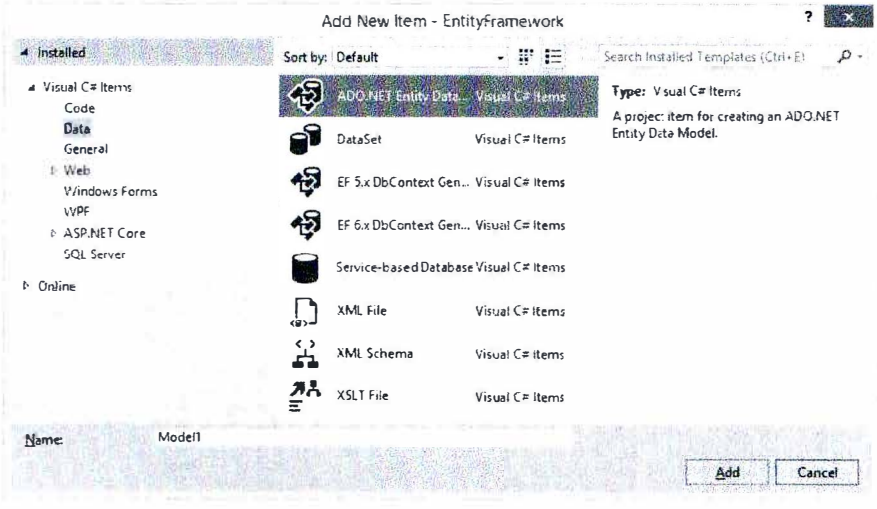


Рис. 24.13. Диалоговое окно Add New Item

3. **Выберите ADO.NET Entity Data Model и введите имя `PartsDatabase`. Щелкните на кнопке Add.**

   Вы увидите диалоговое окно Entity Data Model Wizard, показанное на рис. 24.14.

4. **Выберите пункт Choose Code First from Database в окне Choose Model Contents и щелкните на кнопке Next.**

   Мастер попросит вас выбрать подключение к базе данных. В этом случае вы должны увидеть всю необходимую информацию, потому что она уже была создана для приложения `AccessData`.

5. **Выберите `AdventureWorks2012_Data.mdf` из выпадающего списка Connection и щелкните на кнопке Next.**

   Если в списке нет `AdventureWorks2012_Data.mdf`, обратитесь к разделу "Подключение к источнику данных". Если вы получаете сообщение с вопросом, хотите ли вы скопировать базу данных в проект, выберите No. Как показано на рис. 24.15, мастер попросит вас выбрать объекты базы данных, которые вы хотите использовать.

6. **Выберите Product и ProductCategory и оставьте имя по умолчанию. Щелкните на кнопке Finish.**

Visual Studio сгенерирует для вас необходимый код. При работе с некоторыми версиями Visual Studio вы увидите канву конструктора классов, но поскольку работа с Class Designer — сложная тема, которая может потребовать целой книги сама по себе, здесь мы ее не рассматриваем.

Рис. 24.14. Выберите метод создания модели Entity Framework

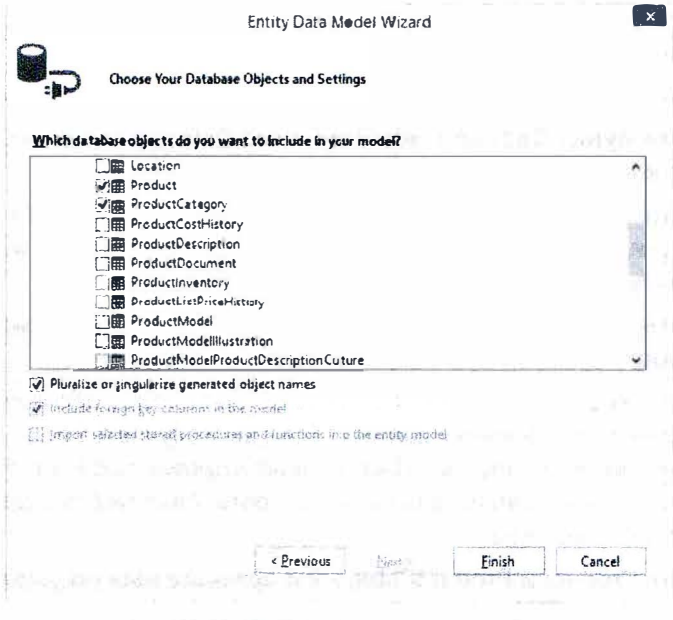


Рис. 24.15. Выбор нескольких таблиц

Написание кода для объектной модели

После того как база данных была аккуратно интегрирована в объектную модель C#, вы можете кодировать с использованием объектов, представленных в этой новой модели. Для начала выполните следующие действия.

1. **Вернитесь к дизайнеру `Form1` и дважды щелкните на `Form1`, чтобы перейти к просмотру кода Code View.**
2. **В обработчике события `Form1_Load()` введите**

   ```
   PartsDatabase part = new PartsDatabase();.
   ```

3. **В следующей строке введите `part` и обратитесь к IntelliSense.**

   Все столбцы таблицы Parts будут представлены в виде свойств класса.

То, что вы получили, — это контекст для дальнейшей работы. Никаких сложных запросов Linq, никакого встроенного SQL, никаких хранимых процедур. Вы можете сделать все, что вам нужно, с помощью полученного объекта.

Глава 25

Рыбалка в потоке

В ЭТОЙ ГЛАВЕ...

- Чтение и запись файлов данных
- Использование классов `Stream`
- Использование конструкции `using`
- Обработка ошибок ввода-вывода

Однажды мне повезло поймать две форели на один крючок в быстром горном потоке в моем родном Колорадо. Это было незабываемое ощущение для одиннадцатилетнего пацана! Ощущения при ловле данных в потоках C# не менее восхитительны, и любой программист обязан их испытать.

Термин *доступ к файлу* (file access) означает сохранение данных на диск и получение их с диска. В этой главе мы рассмотрим основные операции ввода-вывода текстовых файлов.

Где водится рыба: файловые потоки

Консольные программы в настоящей книге в большинстве случаев получают входные данные с консоли и выводят результат работы на консоль. Программы, встречающиеся за пределами книги, как правило, работают с файлами. Вероятность встретить в реальном мире программу, не работающую с файлами, сопоставима с вероятностью встретить в академическом институте рекламу казино в Ницце.

Классы для работы с файлами определены в пространстве имен `System.IO`. Базовым классом для файлового ввода-вывода является класс `FileStream`. Для работы с файлом программист должен его открыть. Команда `open` подготавливает файл к работе и возвращает его дескриптор. Обычно дескриптор — это просто число, которое используется всякий раз при чтении из файла или записи в него.

Потоки

Язык C# использует более интуитивный подход, связывая каждый файл с объектом класса `FileStream`. Конструктор `FileStream` открывает файл и работает с дескриптором файла. Методы `FileStream` осуществляют файловый ввод-вывод.

СОВЕТ

Класс `FileStream` не просто осуществляет файловый ввод-вывод; он в состоянии удовлетворить 90% ваших нужд, связанных с операциями файлового ввода-вывода. Это основной класс, рассматриваемый в данной главе.

Концепция *потока* (stream) является фундаментальной концепцией ввода-вывода в C#. Представьте себе карнавальное шествие, которое "течет" мимо вас: проходят клоуны, жонглеры, проводят пару лошадей, идет труппа объектов `Customer`, объект `BankAccount` и т.д. Можно рассматривать файл как поток байтов (или символов, или строк), очень похожий на такое шествие. Данные "текут" в программу и из нее.

Классы .NET, используемые в C#, включают абстрактный базовый класс `Stream` и ряд подклассов, предназначенных для работы с файлами на диске, по сети или в памяти. Одни классы потоков специализируются на шифровке и расшифровке данных, другие позволяют ускорить операции ввода-вывода, которые могут быть очень медленными при использовании других потоков... Кроме того, вы можете сами дописывать подклассы базового класса `Stream`, если у вас есть свои идеи о том, какие новые виды потоков могут быть полезны вашим программам (но должен предупредить вас, что создание подкласса `Stream` — дело не из легких).

Читатели и писатели

Класс `FileStream` — это, пожалуй, наиболее часто используемый класс потока, представляющий собой простой базовый класс. Открыть файл, закрыть, прочесть или записать блок байтов — вот и все, что он, собственно, умеет. Но чтение и запись файлов на уровне байтов требует большого количества работы, чего я стараюсь избегать. К счастью, библиотека классов .NET вводит

понятия "читателей" и "писателей". Объекты этих типов существенно упрощают файловый (и прочий) ввод-вывод.

Создавая новый *читатель* (одного из доступных типов), вы связываете с ним потоковый объект. Для читателя не важно, связан ли поток с файлом, блоком памяти, местоположением в сети или Миссисипи. Читатель запрашивает входную информацию из потока, который получает ее... словом, откуда-то получает. Использование *писателей* практически аналогично, за исключением того, что вы не запрашиваете входную информацию, а передаете выходную, которую поток отправляет в определенное место, которое часто (но не всегда) является файлом. Пространство имен `System.IO` содержит классы-оболочки для `FileStream` (или иных потоков), которые упрощают доступ.

- **`TextReader/TextWriter`** — пара абстрактных классов для чтения символов (текста). Эти классы предоставляются в двух видах (наборах подклассов): `StringReader/StringWriter` и `StreamReader/StreamWriter`.


ЗАПОМНИ!

Поскольку `TextReader` и `TextWriter` — абстрактные классы, для реальной работы используется одна из пар их подклассов, обычно `StreamReader/StreamWriter`. О том, что такое абстрактные классы, рассказывалось в части 2, "Объектно-ориентированное программирование на C#".

- `StreamReader/StreamWriter` — более интеллектуальные классы чтения и записи текста. Это конкретные классы, которые можно использовать для чтения и записи непосредственно. Например, класс `StreamWriter` имеет метод `WriteLine()`, очень похожий на метод класса `Console`. `StreamReader` имеет соответствующий метод `ReadLine()` и очень удобный метод `ReadToEnd()`, собирающий весь текстовый файл в одну группу и возвращающий считанные символы как строку `string` (которую вы можете затем использовать с классом `StringReader`, циклом `foreach` и т.п.). Классы имеют различные конструкторы, о которых можно прочесть в справочной системе. С применением `StreamReader` и `StreamWriter` в деле вы встретитесь в следующих двух разделах.

Одна очень приятная особенность классов читателей/писателей, таких как `StreamReader` и `StreamWriter`, в том, что их можно использовать с любым типом потоков. Это делает чтение и запись `MemoryStream` не сложнее, чем чтение и запись одного из подвидов `FileStream` (`MemoryStream` будет рассмотрен немного позже в данной главе). Другие пары читателей/писателей вы найдете в разделе "Еще о читателях и писателях".

В следующих разделах будут рассмотрены программы `FileWrite` и `FileRead`, которые демонстрируют способы использования упомянутых классов читателей и писателей для текстового ввода-вывода.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

АСИНХРОННЫЙ ВВОД-ВЫВОД: ЕСТЬ ЛИ ЧТО-ТО ХУЖЕ ОЖИДАНИЯ?

Обычно программа ожидает завершения ее запроса на ввод-вывод и только затем продолжает выполнение. Вызовите метод `read()`, и в общем случае вы не получите управление назад до тех пор, пока данные из файла не будут считаны. Такой способ работы называется *синхронным вводом-выводом*.

Классы C# `System.IO` поддерживают также *асинхронный* ввод-вывод. При использовании асинхронного ввода-вывода вызов `read()` тут же вернет управление программе, позволяя ей заниматься чем-то еще, пока ее запрос на чтение данных из файла выполняется в фоновом режиме. Программа может проверить флаг выполнения запроса, чтобы узнать, завершено ли его выполнение.

Это чем-то напоминает варианты приготовления гамбургеров. При синхронном изготовлении вы нарезаете мясо и жарите его, после чего нарезаете лук и выполняете все остальные действия по приготовлению гамбургера. При асинхронном приготовлении вы начинаете жарить мясо и, поглядывая на него, тут же, не дожидаясь готовности мяса, режете лук и делаете все остальное.

Асинхронный ввод-вывод может существенно повысить производительность программы, но при этом вносит дополнительный уровень сложности.

Использование `StreamWriter`

Программы генерируют два вида вывода.

- **Бинарный**. Некоторые программы пишут блоки данных в виде байтов в чисто бинарном формате. Этот тип вывода полезен для эффективного сохранения объектов (например, файл объектов `Student`, которые сохраняются между запусками программы в файле на диске).

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Сложным примером бинарного ввода-вывода может служить сохранение групп объектов, ссылающихся друг на друга (с использованием отношения СОДЕРЖИТ). Запись объекта на диск включает запись идентифицирующей его информации (чтобы его тип мог быть восстановлен при чтении) и запись каждого из данных-членов, некоторые из которых могут быть ссылками на связанные объекты, каждый со своей идентифицирующей информацией и данными-членами. Сохранение объектов таким образом называется *сериализацией* (serialization).

» **Текстовый**. Большинство программ читает и записывает информацию в виде текста, который может читать человек. Классы `StreamWriter` и `StreamReader` являются наиболее гибкими для работы с данными в таком виде.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Данные в удобном для чтения человеком виде ранее назывались ASCII-строками, а сейчас — ANSI-строками. Эти два термина указывают названия организаций по стандартизации, которые определяют соответствующие стандарты. Однако кодировка ANSI работает только с латинским алфавитом и не имеет кириллических символов, символов иврита, арабского языка или хинди, не говоря уже о такой экзотике, как корейские, японские или китайские иероглифы. Гораздо более гибким является стандарт Unicode, который включает ANSI-символы как свою начальную часть, а кроме них — массу других алфавитов, включая все перечисленные выше. Unicode имеет несколько форматов, именуемых *кодировками*; форматом по умолчанию для C# является UTF8. (Дополнительную информацию о кодировках вы можете найти по адресу `http://unicodebook.readthedocs.io/unicode_encodings.html`.)

Пример использования потока

Приведенная далее программа `FileWrite` считывает строки данных с консоли и записывает их в выбранный пользователем файл.

```
// FileWrite - запись ввода с консоли в текстовый файл
using System;
using System.IO;

namespace FileWrite
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Получение имени файла от пользователя. Цикл while
            // позволяет продолжать попытки с разными именами
            // файлов до тех пор, пока файл не будет успешно
            // открыт
            StreamWriter sw = null;
            string fileName = "";

            while (true)
            {
                try
                {
                    // Ввод имени файла (для выхода из программы
                    // просто нажмите <Enter>)
                    Console.Write("Введите имя файла "
```

```
                    + "(пустое имя для завершения):");
            fileName = Console.ReadLine();

            if (fileName.Length == 0)
            {
                // Имени файла нет - выходим из цикла
                break;
            }

            // Для упрощения цикла задача разделена на
            // подзадачи
            // Вызов метода для создания StreamWriter.
            sw = PrepareTheStreamWriter(fileName);
            // Построчное чтение данных с выводом каждой
            // строки в FileStream, открытый для записи
            ReadAndWriteLines(sw);
            // Запись выполнена, закрываем созданный файл.
            sw.Close(); // Очень важный момент! Этот вызов
            // закрывает и сам файл!
            sw = null;  // Передаем объект сборщику мусора
        }
        catch (IOException ioErr)
        {
            // Произошла ошибка при работе с файлом - о ней
            // надо сообщить пользователю вместе с полным
            // именем файла
            // Класс каталога
            string dir = Directory.GetCurrentDirectory();
            // Класс System.IO.Path
            string path = Path.Combine(dir, fileName);
            Console.WriteLine("Ошибка с файлом {0}", path);
            // Вывод сообщения об ошибке из исключения.
            Console.WriteLine(ioErr.Message);
        }
    }

    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}

// GetWriterForFile - создание StreamWriter для
// записи в конкретный файл
private static StreamWriter
GetWriterForFile(string fileName)
{
    StreamWriter sw;
    // Открываем файл для записи; если файл уже
    // существует, генерируем исключение:
    // FileMode.CreateNew - для создания файла, если он
    // еще не существует, и генерации исключения при
    // наличии такого файла; FileMode.Append для создания
```

```
            // нового файла или добавления данных к существующему
            // файлу; FileMode.Create для создания нового файла
            // или урезания уже имеющегося до нулевого размера.
            // Возможные варианты FileAccess: FileAccess.Read,
            // FileAccess.Write, FileAccess.ReadWrite
            FileStream fs = File.Open(fileName,
                                      FileMode.CreateNew,
                                      FileAccess.Write);
            // Генерируем файловый поток с UTF8-символами (по
            // умолчанию второй параметр дает UTF8, так что он
            // может быть опущен)
            sw = new StreamWriter(fs, System.Text.Encoding.UTF8);
            return sw;
        }

        // WriteFileFromConsole - чтение строк текста с консоли
        // и вывод их в файл.
        private static void
        WriteFileFromConsole(StreamWriter sw)
        {
            Console.WriteLine("Введите текст " +
                              "(пустую строку для выхода)");

            while (true)
            {
                // Считывание очередной строки с консоли; выход,
                // если это пустая строка.
                string input = Console.ReadLine();

                if (input.Length == 0)
                {
                    break;
                }

                // Запись только что считанной строки в файл.
                sw.WriteLine(input);
                // Цикл для считывания и записи очередной строки.
            }
        }
    }
}
```

Программа `FileWrite` использует пространства имен `System.IO` и `System`. Пространство имен `System.IO` содержит классы для файлового ввода-вывода.

Как это работает

Программа начинает работу с функции `Main()`, которая включает цикл `while`, содержащий `try`-блок. В этом нет ничего необычного для программ, работающих с файлами. Размещайте всю работу с файлами в `try`-блоке. Файловому вводу-выводу свойственны ошибки, такие как отсутствие файлов или

каталогов, неверные пути и т.п. Вы уже знаете, как работать с исключениями, из части 1, "Основы программирования на C#".

Цикл `while` служит двум следующим целям.

- Позволяет программе вернуться и повторить попытку в случае, если произошла ошибка ввода-вывода. Например, если демонстрационная программа не может найти файл, который планирует читать пользователь, она может запросить у него имя файла еще раз, а не просто оставить его с сообщением об ошибке.
- Команда `break` в программе переносит вас за `try`-блок, тем самым предоставляя удобный механизм для выхода из функции или программы. Не забывайте о том, что `break` работает только в пределах цикла, в котором вызвана эта команда (а если забыли, перечитайте еще раз главу 5, "Управление потоком выполнения").

Демонстрационная программа `FileWrite` считывает имя создаваемого файла с консоли. Программа прекращает работу путем выхода из цикла `while` с помощью команды `break`, если пользователь вводит пустое имя файла. Ключевым моментом программы является вызов метода `GetWriterForFile()`, в котором главными являются строки

```
FileStream fs = File.Open(fileName,
                          FileMode.CreateNew,
                          FileAccess.Write);
// ...
sw = new StreamWriter(fs, System.Text.Encoding.UTF8);
```

В первой строке программа создает объект `FileStream`, который представляет выходной файл на диске. Конструктор `FileStream`, использованный в данном примере, получает три следующих аргумента.

- **Имя файла.** Это просто имя файла, который следует открыть. Простое имя файла наподобие `filename.txt` предполагает, что файл находится в текущем каталоге (для демонстрационной программы `FileWrite` это подкаталог `\bin\Debug` в каталоге проекта; словом, это каталог, в котором находится сам `.EXE`-файл). Имя файла, начинающееся с обратной косой черты, наподобие `\directory\filename.txt`, рассматривается как полный путь на локальной машине. Имя файла, начинающееся с двух обратных косых черт (например `\\machine\directory\filename.txt`), указывает файл, расположенный на другой машине в вашей сети. Кодировка имени файла — существенно более сложный вопрос, выходящий за рамки данной книги.
- **Режим работы с файлом.** Этот аргумент определяет, что вы намерены делать с файлом. Основными режимами работы с файлом

для записи являются создание (`CreateNew`), добавление к файлу (`Append`) и перезапись (`Create`). `CreateNew` создает новый файл, но генерирует исключение `IOException`, если такой файл уже существует. Простой режим `Create` создает файл, если он отсутствует, но если он есть, то просто перезаписывает его. И наконец `Append` создает файл, если он не существует, но если он есть, открывает его для дописывания информации в конец файла.

» **Тип доступа.** Файл может быть открыт для чтения, записи или для обеих операций.

СОВЕТ

Класс `FileStream` имеет ряд конструкторов, у каждого из которых один или оба аргумента, отвечающие за режим открытия и тип доступа, имеют значения по умолчанию. Однако, по моему скромному мнению, вы должны указывать эти аргументы явно, поскольку это существенно повышает понятность программы. Поверьте, это хороший совет — значения по умолчанию могут быть удобны для программиста, но не для того, кто будет читать его код.

В следующей строке метода `GetWriterForFile()` программа "оборачивает" вновь открытый файловый объект `FileStream` в объект `StreamWriter`. Класс `StreamWriter` служит оберткой для объекта `FileStream`, которая предоставляет набор методов для работы с текстом. Метод возвращает созданный объект `StreamWriter`.

Первый аргумент конструктора `StreamWriter` — объект `FileStream`. Второй аргумент указывает используемую кодировку. Кодировка по умолчанию — UTF8.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Вы не должны указывать кодировку при *чтении* файла. Дело в том, что `StreamWriter` записывает тип применяемой кодировки в первых трех байтах файла. `StreamReader` считывает эти три байта при открытии файла и определяет тип используемой кодировки. Сокрытие такого рода деталей представляет собой одно из преимуществ хорошей библиотеки.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

ЗАВЕРНИТЕ РЫБКУ В ГАЗЕТКУ

"Завертывание" одного класса в другой представляет собой полезный и распространенный шаблон в программировании — `StreamWriter` "обернут" вокруг другого класса, `FileStream` (т.е. содержит ссылку на него), и расширяет интерфейс `FileStream` некоторыми нужными для облегчения жизни методами. Методы `StreamWriter` *делеги-*

руют функциональность (попросту говоря, вызывают) методы внутреннего объекта `FileStream`. Это — рассматривавшееся в части 2, "Объектно-ориентированное программирование на C#", отношение СОДЕРЖИТ, так что всякий раз, когда вы его используете, вы заворачиваете один класс в другой. Таким образом, вы говорите оболочке `StreamWriter`, что надо сделать, а она переводит ваши простые команды в более сложные, необходимые для обернутого класса `FileStream`. Класс `StreamWriter` передает эти транслированные команды классу `FileStream` для выполнения. "Завертывание" — мощный и часто используемый метод программирования. Класс "обертки" `Wrapper` имеет примерно следующий вид:

```
class Wrapper
{
    private Wrapped _wrapped;
    public Wrapper(Wrapped w)
    {
        _w = w; // Теперь у Wrapper есть ссылка на Wrapped.
    }
}
```

В этом примере я использовал конструктор класса `Wrapper` для указания завертываемого объекта, позволив вызывающему методу передать его как параметр. Это можно сделать при помощи специального метода `SetWrapped()` или иным способом, включая создание оборачиваемого объекта в конструкторе класса.

Можно также обернуть один *метод* вокруг другого, примерно так:

```
void WrapperMethod()
{
    _wrapped.DoSomething();
}
```

В этом примере класс метода `WrapperMethod()` СОДЕРЖИТ ссылку на некоторый объект `_wrapped`. Другими словами, класс "оборачивает" этот объект. Метод `WrapperMethod()` просто делегирует свою функциональность — полностью или частично — методу `DoSomething()` объекта `_wrapped`.

Завертывание можно представить как способ преобразования одной модели в другую. Завернутый элемент может быть таким сложным, что вы хотели бы предоставить более простую его версию; или, может быть, у него неудобный интерфейс, который бы вы хотели превратить в более подходящий. Вообще говоря, завертывание иллюстрирует проектный шаблон Adapter (который вы можете поискать в Google). Вы можете увидеть его в отношениях классов `StreamWriter` и `FileStream`. В ряде случаев можно обернуть один поток вокруг другого для того, чтобы преобразовать один вид потока в другой.

Наконец-то мы пишем!

После настройки `StreamWriter` программа `FileWrite` считывает входные строки с консоли (этот код находится в методе `WriteFileFromConsole()`, вызываемом из метода `Main()`). Программа завершает работу после ввода пользователем пустой строки, но до этого все введенное пользователем выводится в файл при помощи метода `WriteLine()` класса `StreamWriter`. И наконец поток закрывается с помощью вызова `sw.Close()`. Это весьма важное действие, поскольку оно закрывает и файл.

СОВЕТ

Обратите внимание на то, что программа обнуляет ссылку `sw` по закрытии `StreamWriter`. Файловый объект становится бесполезным после того, как файл закрыт. Правила хорошего тона требуют обнулять ссылки после того, как они становятся недействительными, так, чтобы обращений к ним больше не было (если вы попытаетесь это сделать, то будет сгенерировано исключение). Закрытие файла и обнуление ссылки позволяет сборщику мусора подобрать ненужную более память, а другим программам — открывать закрытый файл.

Блок `catch` напоминает футбольного вратаря: он стоит здесь для того, чтобы ловить все исключения, которые могут быть сгенерированы в программе. Он выводит сообщение об ошибке, включая имя вызвавшего ее файла. Однако выводится не просто имя файла, а его полное имя, включая путь к нему. Это делается посредством класса `Directory`, который позволяет получить текущий каталог и добавить его перед введенным именем файла с использованием метода `Path.Combine()` (`Path` — класс, разработанный для работы с информацией о путях, а `Directory` предоставляет свойства и методы для работы с каталогами).

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

РАБОТА С СИМВОЛАМИ ПУТИ

Метод `Combine()` достаточно интеллектуален, чтобы разобраться, что для файла наподобие `c:\test.txt` `Path()` не является текущим каталогом. `Path.Combine()` представляет также наиболее безопасный путь, гарантирующий корректное объединение двух частей пути, включая символ-разделитель (\) между ними.

В Windows символ-разделитель пути — \, но можно использовать и символ-разделитель Linux — /. Вы можете получить корректный разделитель для операционной системы, под управлением которой запущена программа, с помощью `Path.DirectorySeparatorChar`. Библиотека .NET Framework изобилует такого рода возможностями, существенно облегчая программистам на C# написание программ, которые должны работать под управлением нескольких операционных систем.

ЗАПОМНИ!

Путь — это полное имя каталога. Например, если имя файла — `c:\user\directory\text.txt`, то его путь — `c:\user\directory`.

Достигнув конца цикла `while`, либо после выполнения `try`-блока, либо после блока `catch`, программа возвращается к началу цикла и позволяет пользователю записать другой файл. Вот как выглядит пример выполнения демонстрационной программы (пользовательский ввод выделен полужирным шрифтом).

```
Введите имя файла (пустое имя для завершения):TestFile1.txt
Введите текст (пустую строку для выхода)
Это какой-то текст
И еще
И еще раз...

Введите имя файла (пустое имя для завершения):TestFile1.txt
Ошибка с файлом C:\C#Programs\FileWrite\bin\Debug\TestFile1.txt
The file already exists.

Введите имя файла (пустое имя для завершения):TestFile2.txt
Введите текст (пустую строку для выхода)
Я ошибся - мне надо было ввести
имя файла TestFile2.

Введите имя файла (пустое имя для завершения):
Нажмите <Enter> для завершения программы...
```

Все отлично работает, пока некоторый текст вводится в файл `TestFile1.txt`. Но при попытке открыть файл `TestFile1.txt` заново программа выводит сообщение `The file already exists` (файл уже существует). Обратите внимание на полный путь к файлу, выводимый вместе с сообщением об ошибке. Если исправить ошибку и ввести имя `TestFile2.txt`, все продолжает отлично работать.

Использование конструкции `using`

Теперь, когда вы увидели `FileStream` и `StreamWriter` в действии, я должен указать на более обычный путь записи потоков в C# — в конструкции `using`:

```
using(<некоторый ресурс>)
{
  // Use the resource.
}
```

Конструкция `using` автоматизирует процесс освобождения ресурсов после использования потока. Когда C# встречает закрывающую фигурную скобку блока `using`, он "сбрасывает" поток и закрывает его. (*Сброс* потока

представляет собой "выталкивание" всех записанных байтов из буфера в файл перед его закрытием. Этот сброс можно представить как выдавливание последних капель потока.) Использование `using` позволяет устранить распространенную ошибку, заключающуюся в забывании сбросить и закрыть файл после записи. Не бросайте открытые файлы как угодно и где попало... Без конструкции `using` требуется написать

```
Stream fileStream = null;
TextWriter writer = null;
try
{
    // Создание и использование потока; затем ...
}
finally
{
    stream.Flush();
    stream.Close();
    stream = null;
}
```

(Обратите внимание на то, что я объявил поток и писатель до `try`-блока, чтобы они были видимы везде в методе. Я также объявил переменные `fileStream` и `writer` с использованием абстрактных базовых классов, а не конкретных типов `FileStream` и `StreamWriter`. Это хорошее решение. Я сделал их равными `null`, чтобы компилятор не стал жаловаться на неинициализированные переменные.) А вот как выглядит предпочтительный способ записи ключевого кода ввода-вывода в примере `FileWrite`:

```
// Подготовка файлового потока.
FileStream fs = File.Open(fileName,
                          FileMode.CreateNew,
                          FileAccess.Write);
// Передача переменной fs конструктору StreamWriter в
// конструкции using.
using (StreamWriter sw = new StreamWriter(fs))
{
    // sw существует только в блоке using, который
    // представляет собой локальную область видимости.

    // Считывание строк с консоли по одной, передача каждой из
    // них объекту FileStream для записи.
    Console.WriteLine("Введите текст " +
                      "(пустую строку для выхода)");
    while (true)
    {
        // Считывание очередной строки с консоли; завершение
        // работы при считывании пустой строки.
        string input = Console.ReadLine();
        if (input.Length == 0)
        {
```

```
            break;
        }
        // Запись только что считанной строки в файл при помощи
        // потока.
        sw.WriteLine(input);
        // Цикл для считывания и записи очередной строки.
    }
} // Здесь sw выходит из области видимости, и fs
  // закрывается.
fs = null;  // Гарантируем невозможность обратиться к
            // закрытому файлу еще раз.
```

Объекты в круглых скобках после ключевого слова `using` представляют собой раздел "захвата ресурса", в котором выделяется один или несколько ресурсов, таких как потоки, читатели–писатели, шрифты и т.д. (Если вы захватываете несколько ресурсов, они должны быть одного типа.) За этим разделом следует охватывающий блок, ограниченный фигурными скобками.

ЗАПОМНИ!

Блок конструкции `using` не является циклом. Он просто определяет локальную область видимости, как и блок `try` или блок метода. (Переменные, определенные в блоке, включая его заголовок, вне блока не существуют. Таким образом, переменная `sw` типа `StreamWriter` невидима вне блока `using`.) Области видимости рассматривались в части 1, "Основы программирования на C#".

В приведенном примере в разделе захвата ресурсов выполняются действия по настройке ресурсов — в нашем случае создается новый объект `StreamWriter`, обернутый вокруг уже существующего `FileStream`. В блоке выполняются все действия во вводу-выводу в файл.

В конце блока `using` C# автоматически сбросит `StreamWriter`, закроет его и `FileStream` и сбросит все байты из памяти на диск. Но это еще не все — окончание блока `using`, кроме того, *освобождает* объект `StreamWriter`.

СОВЕТ

Работу с потоками имеет смысл всегда размещать в блоке конструкции `using`. Так, размещение в этом блоке объектов `StreamWriter` или `StreamReader` действует так же, как и размещение использования писателя или читателя в блоке обработки исключений `try/finally`. Фактически компилятор транслирует блок `using` в такой же код, как и при трансляции `try/finally`, что гарантирует корректное освобождение всех захваченных ресурсов:

```
try
{
    // Выделение ресурсов и их использование.
}
```

```
finally
{
    // Закрытие и освобождение захваченных ресурсов.
}
```


ВНИМАНИЕ!

За пределами блока `using` не только больше не существует объект `StreamWriter`, но и нельзя обращаться к объекту `FileStream`. *Переменная* `fs` все еще существует — в предположении, что поток был создан вне конструкции `using`, а не "на лету", как в приведенном фрагменте:

```
using(StreamWriter sw = new StreamWriter(new FileStream(...)) ...
```

Однако сброс и закрытие писателя сбрасывает и закрывает и сам файл. Если вы попытаетесь выполнять операции с потоком, то получите исключение, гласящее, что вы не можете работать с закрытым объектом. Обратите внимание на то, что в коде `FileWrite` ранее в этом разделе я обнулял объект `fs` после блока `using`, чтобы гарантировать невозможность повторного использования `fs`. После этого объект `FileStream` передается сборщику мусора.

Конечно же, файл, который вы записали на диск, продолжает существовать. Если вам надо будет работать с ним еще раз, для этого следует создать и открыть новый файловый поток.


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Конструкция `using` обеспечивает освобождение ресурсов для объектов, которые реализуют интерфейс `IDisposable` (об интерфейсах рассказывается в части 2, "Объектно-ориентированное программирование на C#"). Конструкция `using` гарантирует вызов метода `Dispose()` объекта. Классы, которые реализуют `IDisposable`, гарантированно имеют такой метод. Интерфейс `IDisposable` в основном предназначен для освобождения ресурсов, не являющихся ресурсами .NET, главным образом ресурсов внешнего мира — операционной системы Windows, таких как дескрипторы файлов и графические ресурсы. Например, класс `FileStream` представляет собой обертку вокруг файлового дескриптора Windows, который должен быть освобожден. (Интерфейс `IDisposable` реализует множество классов и структур; ваши собственные классы также могут это делать.)

В этой книге я не буду углубляться в дебри интерфейса `IDisposable`, но по мере получения опыта работы с C# вы обязательно должны поближе с ним познакомиться. Корректная его реализация работает с недетерминированной сборкой мусора и может быть весьма сложной. Конструкция `using` используется с классами и структурами, которые реализуют интерфейс `IDisposable`; реализует ли конкретный класс этот интерфейс, можно узнать из справочной системы. *Эта конструкция не работает для произвольных типов объектов.*

Примечание. Встроенные типы C# — `int`, `double`, `char` и др. — ***не*** реализуют интерфейс `IDisposable`. Класс `TextWriter`, базовый класс для `StreamWriter`, реализует этот интерфейс. В справочной системе это выглядит следующим образом:

```
public abstract class TextWriter : MarshalByRefObject, IDisposable
```

Если вы не знаете, реализует ли данный класс или структура интерфейс `IDisposable`, прежде чем использовать его, проконсультируйтесь со справочной системой.

Использование `StreamReader`

Запись файла — дело стоящее, но совершенно бесполезное, если вы не можете позже прочесть записанное. Приведенная ниже демонстрационная программа `FileRead` считывает текстовый файл, например, созданный демонстрационной программой `FileWrite` или программой Блокнот, т.е. это обращенная программа `FileWrite` (замечу, что в ней я решил не использовать конструкцию `using`):

```
// FileRead - чтение текстового файла и вывод считанной
// информации на консоль.
using System;
using System.IO;

namespace FileRead
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Нам нужен объект читателя файла.
            StreamReader sr = null;
            string fileName = "";

            try
            {
                // Получение имени файла от пользователя.
                sr = GetReaderForFile(fileName);
                // Чтение содержимого файла.
                ReadFileToConsole(sr);
            }
            catch (IOException ioErr)
            {
                //TODO: перед тем как выпускать окончательную
                // версию, следует заменить сообщение более
                // понятным.
                Console.WriteLine("{0}\n\n", ioErr.Message);
            }
```

```
    finally  // Освобождение ресурсов.
    {
        if (sr != null) // Защита от попытки вызвать Close()
        {
            // для объекта null.
            sr.Close();    // Сброс файла на диск.
            sr = null;
        }
    }
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}

// GetReaderForFile - открываем файл и возвращаем
// связанный с ним StreamReader.
private static StreamReader
GetReaderForFile(string fileName)
{
    StreamReader sr;
    // Ввод имени входного файла.
    Console.Write("Введите имя считываемого " +
                  "текстового файла:");
    fileName = Console.ReadLine();

    // Пользователь ничего не ввел; генерируем исключение
    // для указания неприемлемости такого ввода.
    if (fileName.Length == 0)
    {
        throw new IOException("Вам надо ввести имя файла.");
    }

    // Получение имени - открытие файлового потока для
    // чтения; не создавайте файл, если его не существует.
    FileStream fs = File.Open(fileName, FileMode.Open,
                              FileAccess.Read);
    // Обертываем StreamReader вокруг потока. Первые три
    // байта файла указывают использованную кодировку (но
    // не язык).
    sr = new StreamReader(fs, true);
    return sr;
}

// ReadFileToConsole - считываем строки из файла,
// представленного sr, и выводим их на консоль.
private static void ReadFileToConsole(StreamReader sr)
{
    Console.WriteLine("\nСодержимое файла:");

    // Считывание по одной строке.
    while (true)
    {
        // Читаем строку.
        string input = sr.ReadLine();
```

```
                // Выход, если больше нечего читать.
                if (input == null)
                {
                    break;
                }

                // Записываем считанное на консоль.
                Console.WriteLine(input);
            }
        }
    }
}
```

ВНИМАНИЕ!

Вспомните, что текущий каталог, используемый программой `FileRead`, — это подкаталог `\bin\Debug` в проекте `FileRead` (не подкаталог `\bin\Debug` в каталоге программы `FileWrite`, где создавался файл, записываемый этой программой `FileWrite` в предыдущем разделе). Перед тем как запустить программу `FileRead` для проверки ее работоспособности, поместите любой обычный текстовый файл (с расширением `.TXT`) в подкаталог `\bin\Debug` каталога `FileRead` и запомните его имя, чтобы вы могли его открыть. Для этого вполне подойдет копия файла `TestFilel.txt`, созданного программой `FileWrite`.

В программе `FileRead` пользователь читает один и только один файл. Пользователь может ввести корректное имя файла для вывода (второго шанса не дано). После того как программа прочтет файл, она завершает свою работу. Если пользователь хочет выбрать второй файл, он должен перезапустить программу. Вы, конечно же, можете организовать работу своей программы совершенно иначе.

Весь серьезный код программы помещен в обработчик исключений. В `try`-блоке вызываются два метода: сначала — для получения объекта `StreamReader` для файла, а затем — для чтения файла и вывода его содержимого на консоль. В случае генерации исключения `catch`-блок выводит сообщение об исключении. Наконец, независимо от того, генерируется исключение или нет, блок `finally` обеспечивает закрытие потока и его файла и обнуление переменой `sr` для того, чтобы сборщик мусора мог освободить неиспользуемую более память (см. часть 2, "Объектно-ориентированное программирование на C#"). Исключения ввода-вывода могут быть сгенерированы в обоих методах, вызываемых в `try`-блоке. Эти исключения в поисках обработчика доходят до метода `Main()` (обработчики исключений в самих этих методах не нужны).

СОВЕТ

Обратите внимание на комментарий `//TODO:` в `catch`-блоке. Это напоминание о том, что сообщение следует сделать более понятным для пользователя перед тем, как окончательно выпускать программу.

Помеченные таким образом комментарии в Visual Studio выводятся в окне Task List. Выберите в этом окне в разворачивающемся списке в верхней левой его части Comments. Двойной щелчок на элементе списка откроет редактор на соответствующей строке исходного текста.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Поскольку переменная `sr` используется в блоке исключения, ее следует изначально установить равной `null`, так как в противном случае компилятор сообщит об использовании неинициализированной переменной в блоке исключения. То же самое относится и к вызову ее метода `Close()`. Но еще лучше переписать программу так, чтобы она использовала конструкцию `using`.

В методе `GetReaderForFile()` программа дает пользователю единственный шанс ввести имя файла. Если пользователь введет пустое имя файла, программа сгенерирует собственное сообщение об ошибке "`Вам надо ввести имя файла.`" Если же имя файла не пустое, оно используется для открытия объекта `FileStream` в режиме для чтения. Вызов `File.Open()` такой же, как и используемый в программе `FileWrite`.

- Первый аргумент метода — имя файла.
- Второй аргумент — режим открытия файла. Режим `FileMode.Open` гласит: "Открыть файл, если он существует, и сгенерировать исключение, если его нет". Другой вариант — `OpenNew`, который создает файл нулевой длины в случае отсутствия последнего.
- Третий аргумент указывает на желание читать из объекта `FileStream`. Другие возможные варианты — `Write` и `ReadWrite`. (Кажется странным открывать файл в программе `FileRead` с использованием режима `Write`, не правда ли?)

Полученный в результате объект `fs` класса `FileStream` оборачивается в объект `sr` класса `StreamReader` для предоставления удобного доступа к текстовому файлу. Объект `StreamReader` в конечном итоге передается в метод `Main()` для дальнейшего использования.

После завершения процесса открытия файла программа `FileRead` вызывает метод `ReadFileToConsole()`, который в цикле считывает строки текста из файла при помощи вызовов метода `ReadLine()`. Программа выводит каждую строку на консоль вызовом `Console.WriteLine()`. Когда программа достигает конца файла, вызов `ReadLine()` возвращает значение `null`. Когда это происходит, метод завершает цикл чтения, а затем осуществляется выход из метода. После этого метод `Main()` закрывает объект и завершает работу программы. (Можно сказать, что считывающая часть программы завернута в цикл `while` внутри метода, который находится в `try`-блоке, обернутом в...)

Блок `catch` в методе `Main()` требуется для того, чтобы сгенерированное исключение не привело к аварийному останову программы. Если программа генерирует исключение, `catch`-блок выводит сообщение и просто игнорирует ошибку. Этот блок `catch` позволяет пользователю узнать, что же произошло, и предупреждает аварийное завершение программы из-за необработанного исключения. Можно переписать программу таким образом, чтобы у пользователя запрашивалось другое имя файла, но данная программа настолько мала, что это не имеет смысла — проще запустить ее заново.


СОВЕТ

Наличие обработчика исключения, который просто перехватывает ошибку, предохраняет программу от аварийного завершения из-за мелкой неприятности. Однако этот метод можно использовать, только если ошибка действительно некритична и не вредит работе программы.

Вот как выглядит пример вывода программы:

```
Введите имя считываемого текстового файла:TestFilex.txt
Could not find file "C:\C#Programs\FileRead\TestFilex.txt".
Нажмите <Enter> для завершения программы...

Введите имя считываемого текстового файла:TestFile1.txt
Содержимое файла:
Это какой-то текст
И еще
И еще раз...
Нажмите <Enter> для завершения программы...
```

Пример чтения произвольных *байтов* из файла (который может быть как текстовым, так и бинарным) показан в программе `LoopThroughFiles` в главе 7, “Работа с коллекциями”. Программа циклически просматривает все файлы в целевом каталоге, читая каждый файл и выводя его содержимое на консоль; очевидно, что она быстро становится утомительной при наличии большого количества файлов. Не стесняйтесь прекратить ее работу нажатием клавиш <Ctrl+C> или щелчком на кнопке закрытия окна консоли. Смотрите обсуждение `BinaryReader` в следующем разделе.

Еще о читателях и писателях

Ранее в этой главе я показал вам классы `StreamReader` и `StreamWriter`, которые, пожалуй, способны удовлетворить подавляющее большинство ваших нужд в файловых операциях ввода-вывода. Однако библиотека .NET предлагает ряд других пар “читатель–писатель”.

» `BinaryReader/BinaryWriter` — пара потоковых классов, которые содержат методы для чтения и записи каждого из типов-значений: `ReadChar()`, `WriteChar()`, `ReadByte()`, `WriteByte()` и т.д. Эти классы полезны для чтения и записи объекта в бинарном (не читаемом человеком) формате, в противоположность текстовому формату. Для работы с бинарными данными можно использовать массив или коллекцию байтов.

Эксперимент: откройте в программе Блокнот файл с расширением `.EXE`. В окне вы можете увидеть читаемые фрагменты текста, но большая часть файла будет выглядеть мусором. Это и есть бинарные данные.

В главе 7, "Работа с коллекциями", имеется упомянутый ранее пример, который читает бинарные данные. В этом примере `BinaryReader` с объектом `FileStream` используется для чтения блоков байтов из файла с последующим выводом на консоль в шестнадцатеричной записи. Несмотря на то что пример оборачивает `FileStream` в более удобный `BinaryReader`, в этом примере с таким же успехом можно было бы использовать `FileStream` непосредственно.

» `StringReader/StringWriter` — простые потоковые классы, которые ограничены чтением и записью строк. Они позволяют рассматривать строку как файл, предоставляя альтернативу доступу к символам строк с помощью записи с использованием квадратных скобок (`[]`), цикла `foreach` или методов класса `String` наподобие `Split()`, `Concatenate()` и `IndexOf()`. Вы считываете и записываете строки почти так же, как и файлы. Этот метод полезен для длинных строк с сотнями или тысячами символов (например, полностью считанный в строку текстовый файл), которые вы хотите обработать вместе. Методы в этих классах аналогичны методам классов `StreamReader` и `StreamWriter`, описываемым далее.

При создании объекта типа `StringReader` вы инициализируете его строкой для чтения. При создании объекта `StringWriter` ему либо передается существующий объект типа `StringBuilder`, либо создается пустой объект такого типа. Получить содержимое внутреннего объекта `StringBuilder` можно при помощи вызова метода `ToString()` класса `StringWriter`.

Всякий раз при чтении из строки (или записи в нее) "указатель положения в файле" перемещается к следующему символу. Таким образом, как и при файловом вводе-выводе, здесь используется понятие "текущая позиция". При чтении 10 символов из тысячесимвольной строки указатель после чтения окажется указывающим на одиннадцатый символ.

Методы этих классов аналогичны описанным методам классов `StreamReader` и `StreamWriter`. Если вы применяли их там, то можете использовать и здесь.

Другие виды потоков

В завершение я должен упомянуть, что файловые потоки — не единственный доступный вид подклассов `Stream`. Поток подклассов `Stream` включает (но не ограничивается ими) классы из приведенного далее списка. Если явно не указано иное, классы находятся в пространстве имен `System.IO`.

- **`FileStream`**. Считывает и записывает файлы на диск.
- **`MemoryStream`**. Управляет чтением и записью данных в блоки памяти. Этот метод иногда применяется при тестировании, чтобы избежать медленной и беспокойной работы с диском, подменив дисковый файл "обманкой" в памяти.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

- **`BufferedStream`**. *Буферизация* представляет собой метод ускорения операций ввода-вывода путем считывания или записи больших блоков данных за один раз. Много операций чтения или записи малого размера означают большое количество медленных обращений к диску. Вместо них можно считывать в буфер сразу большие блоки данных, а уже затем выбирать отдельные байты из буфера, что будет существенно быстрее работы с диском. Когда в буфере `BufferedStream` данные исчерпываются, он считывает новую порцию данных, вплоть до всего файла целиком. Буферизация записи выполняется аналогично.

  Класс `FileStream` автоматически буферизует свои операции, так что `BufferedStream` предназначается для особых случаев наподобие работы с `NetworkStream`, который читает и пишет байты по сети. В этом случае `BufferedStream` обертывается вокруг `NetworkStream`, и когда вы пишете данные в `BufferedStream`, они записываются в обернутый `NetworkStream`.
- **`NetworkStream`**. Управляет чтением и записью данных по сети. Класс `NetworkStream` находится в пространстве имен `System.Net.Sockets`, поскольку для соединения по сети он использует сокеты.
- **`UnmanagedMemoryStream`**. Позволяет читать и писать данные в "неуправляемые" блоки памяти. Под *неуправляемостью* в данном случае подразумевается отсутствие управления со стороны .NET и сборщика мусора.
- **`CryptoStream`**. Находится в пространстве имен `System.Security.Cryptography`. Этот класс потока предназначен для передачи данных с шифрованием.

Глава 26

Доступ к Интернету

В ЭТОЙ ГЛАВЕ...

- Экскурсия по пространству имен `System.Net`
- Встроенные инструменты для работы с сетью
- Работа с сетевыми инструментами

Причиной, по которой Microsoft пришлось создавать .NET Framework, было отсутствие в существующей инфраструктуре возможностей для взаимодействия с Интернетом. Объектная модель компонентов (Component Object Model — COM) просто не в состоянии работать с Интернетом. Интернет работает не так, как большинство платформ, таких как персональные компьютеры. Интернет построен на протоколах — точно определенных и согласованных способах обеспечения работы таких средств, как электронная почта и передача файлов. Среда Microsoft до 2002 года явно не справлялась с этим.

Как вы могли видеть в этой книге, каркас .NET изначально разрабатывался с учетом Интернета и сетей в целом. Не удивительно, что это особенно ярко видно в пространстве имен `System.Net`. Здесь первое место занимает Интернет, а веб-инструменты представлены девятью классами в этом пространстве имен.

В версии .NET 4.7 (которая поставляется вместе с Visual Studio 2017) добавлено еще больше функциональных возможностей для работы с Интернетом. Хотя в версиях 1.*x* основное внимание уделялось инструментам, используемым для создания других инструментов (низкоуровневых функций), теперь каркас содержит полезные функции, такие как веб, электронная почта и FTP.

SSL (Secure Sockets Layer) — транспортная безопасность Интернета — в этой версии намного проще, как и FTP и почта, которые ранее требовали применения других, более сложных в использовании классов.

`System.Net` — большое пространство имен, поиск в котором может быть трудным. В этой главе обсуждаются часто выполняемые задачи и показаны основы работы с Интернетом. Работа в сети составляет большую часть .NET Framework, и все соответствующие функциональные возможности находятся в указанном пространстве имен. На эту тему может быть (и была) написана не одна книга. В качестве введения в сетевые возможности C# в этой главе представлены следующие функции:

- получение файлов из сети;
- отправка электронной почты;
- регистрация пересылаемой информации;
- проверка состояния сетевого окружения вашего приложения.

Имейте в виду, что сокеты и IPv6, как и другие современные протоколы Интернета, достаточно важны, но большинство разработчиков в настоящее время не используют их каждый день. В этой главе рассказывается о тех частях пространства имен, которые вы будете использовать в ежедневной работе. Так что, как и в любой теме этой книги — имеется большое количество информации о пространстве имен `System.Net`, не освещенной здесь.

Знакомство с `System.Net`

Пространство имен `System.Net` содержит множество классов, которые могут смутить незрелого программиста своим количеством при просмотре документации, но которые имеют большой смысл при использовании в приложении. Пространство имен устраняет всю сложность работы с различными протоколами, используемыми в Интернете.

Существует более 2000 RFC для протоколов Интернета (RFC (Request For Comments — запрос комментариев) представляет собой документ, который отправляется в орган по стандартизации для проверки коллегами, прежде чем он станет стандартом). Если вам нужно изучить все необходимые для работы приложения RFC по отдельности, вы никогда не завершите свой проект. Пространство имен `System.Net` позволяет сделать сетевое кодирование менее болезненным.

`System.Net` предназначено не только для веб-проектов. Как и все остальное в библиотеке базовых классов, вы можете использовать `System.Net` с проектами всех видов. Вы можете:

- получать информацию из веб-страниц в Интернете и использовать ее в своих программах;
- пересылать файлы с использованием FTP;
- легко отправлять электронную почту;
- использовать современные сетевые структуры;
- обеспечивать безопасность соединений в Интернете с использованием протокола SSL.

Если вам нужно проверить подключение компьютера из приложения Windows, вы можете использовать `System.Net`. Если вам нужно создать класс, который будет загружать файл с веб-сайта, то `System.Net` — именно то пространство имен, которое вам нужно. То, что большинство классов относятся к работе с Интернетом, не означает, что их могут использовать только веб-приложения. В этом — магия `System.Net`. Любое приложение может быть подключенным к сети. В то время как некоторые части данного пространства имен предназначены для облегчения разработки веб-приложений, в целом пространство имен `System.Net` предназначено для того, чтобы любое приложение могло работать с сетями, соответствующими веб-стандартам (включая внутренние сети организаций).

Как сетевые классы вписываются в каркас

Пространство имен `System.Net` содержит большое количество классов (см. `https://msdn.microsoft.com/en-us/library/system.net(v=vs.110).aspx`) и меньшие пространства имен (`https://msdn.microsoft.com/en-us/library/gg145039(v=vs.110).aspx`). Количество классов и пространств имен увеличивается с каждой версией .NET Framework, поэтому следует следить за каждым обновлением, чтобы знать, что нового появляется в каркасе. Разнообразие возможностей может показаться подавляющим. Тем не менее при внимательном исследовании можно увидеть единую схему.

Классы хорошо именованы, и можно заметить, что каждый протокол включает несколько классов.

- `Authentication` и `Authorization`: классы, обеспечивающие безопасность.
- `Cookie`: класс, управляющий cookie-файлами, используемыми веб-браузерами и страницами ASP.NET.
- `DNS` (Domain Name Services — служба доменных имен): классы помогают разрешению доменных имен в IP-адреса.

- `Download`: класс используется для получения файлов с серверов.
- `EndPoint`: класс, помогающий определить сетевой узел.
- `FileWeb`: набор классов, описывающий сетевые файловые сервера как локальные классы.
- `FtpWeb`: класс простой реализации протокола передачи файлов (File Transfer Protocol).
- `Http` (HyperText Transfer Protocol — протокол передачи гипертекста): класс, предоставляющий протокол связи с веб-серверами.
- `IP` (Internet Protocol — протокол Интернета): класс, помогающий определить конечные точки сети, связанные с Интернетом.
- `IrDA`: класс для работы с инфракрасными портами.
- `NetworkCredential`: еще один класс обеспечения безопасности.
- `Service`: вспомогательный класс для управления сетевыми подключениями.
- `Socket`: класс для работы с примитивами сетевых подключений.
- `Upload`: набор классов для загрузки информации в Интернет.
- `Web`: классы для помощи в работе с WWW — реализации специализированных классов.

Этот список столь обширный, потому что одни классы строятся на основе других. Классы `EndPoint` используются классами `Socket` для определения некоторых определенных особенностей сети, а классы `IP` обеспечивают их работу в Интернете. `Web`-классы специфичны для работы в WWW. Зачастую трудно понять, какие классы когда нужны. Однако большинство повседневно используемых функций инкапсулированы в семь подпространств имен в пространстве имен `System.Net`.

- `Cache`: множество перечислений, управляющих кешированием браузеров и сетей.
- `Configuration`: обеспечение доступа к свойствам, необходимым для настройки работы множества других классов `System.Net`.
- `Mail`: облегчение отправки электронной почты через Интернет.
- `Mime`: связь файловых вложений с пространством имен `Mail` с использованием стандарта MIME.
- `NetworkInformation`: получение детальной информации о сетевом окружении приложения.
- `Security`: реализация сетевой безопасности классами `System.Net`.
- `Sockets`: базовые сетевые подключения в Windows.

Использование пространства имен System.Net

Пространство имен System.Net *ориентировано на код*, а это означает, что только немногие реализации предназначены для работы в качестве пользовательского интерфейса. Почти все, что вы делаете с этими классами, происходит "за сценой". У вас есть только несколько перетаскиваемых с помощью мыши пользовательских управляющих элементов — пространство имен System.Net используется в представлении Code. Чтобы продемонстрировать этот факт, примеры оставшейся части главы создают приложение Windows Forms, которое выполняет следующее:

- проверяет состояние сети;
- получает определенный файл из Интернета;
- отправляет его почтой на определенный адрес(или адреса);
- записывает информацию о транзакции.

Это не столь уж незначительный набор действий. На самом деле в версиях C# 1.0 и 1.1 написать такую программу было бы очень сложно. Одна из основных целей пространства имен System.Net и состоит в том, чтобы облегчить выполнение распространенных сетевых задач. Вы можете начать с загрузки кода примера или создания нового проекта, следуя инструкциям в следующих разделах.

Проверка состояния сети

Сначала вам нужно проинформировать пользователя о подключении к сети. Соответствующее приложение создается с помощью следующих действий.

1. **Выберите пункт меню создания нового проекта File⇨New⇨Project.**

   Вы увидите диалоговое окно нового проекта New Project.
2. **Выберите на левой панели Visual C#\Windows Classic Desktop.**
3. **Выберите шаблон Windows Forms App на центральной панели.**
4. **Введите NetworkTools в поле Name и щелкните на кнопке OK.**

   Visual Studio создаст приложение Windows Form и выведет окно, в котором вы можете начать добавлять на форму управляющие элементы.
5. **Добавьте управляющий элемент StatusStrip в нижнюю левую часть формы, перетащив его из группы Menus&Toolbars набора инструментов Toolbox.**

   Управляющий элемент StatusStrip автоматически займет всю нижнюю часть формы.

6. **Выберите SmartTag в левой части StatusStrip и добавьте управляющий элемент StatusLabel.**

   На рис. 26.1 показаны элементы `SmartTag`, появляющиеся после щелчка на направленной вниз стрелке.

Рис. 26.1. Выбор управляющего элемента StatusLabel

7. **Дважды щелкните на форме.**

   Visual Studio создаст метод `Form1_Load()` и откроет редактор кода.

8. **Добавьте в начало кода строку `using System.NET.NetworkInformation;`.**

9. Добавьте код из приведенного далее листинга для проверки доступности сети и вывода соответствующей информации в строке состояния.

```
using System;
using System.Windows.Forms;
using System.Net.NetworkInformation;
namespace NetworkTools
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            if (NetworkInterface.GetIsNetworkAvailable())
            {
                toolStripStatusLabel1.Text = "Connected";
            }
```

```
            else
            {
                toolStripStatusLabel1.Text = "Disconnected";
            }
        }
    }
}
```

Это все, что нужно сделать. Класс `NetworkInformation` содержит пакет информации о состоянии сети, текущих IP-адресах, используемом компьютером шлюзе и т.д.


СОВЕТ

Имейте в виду, что класс `NetworkInformation` будет работать только на локальном компьютере. Используя этот класс в приложении ASP.NET Web Forms, вы получите информацию о сервере.

Загрузка файла из Интернета

Получить файл из Интернета можно несколькими способами, и одним из наиболее распространенных является использование протокола передачи файлов FTP. Протокол FTP предпочтителен, потому что он безопасен и поддерживается во многих системах. Чтобы создать приложение, использующее FTP, начните с примера из предыдущего раздела и выполните следующие действия.

1. **Перетащите управляющий элемент Button из Toolbox на форму.**
2. **Дважды щелкните на нем.**

   Visual Studio создаст метод `button1_Click()` и откроет его код в редакторе.
3. **Добавьте в верхней части кода следующие строки:**

```
using System.Net;
using System.IO;
```

4. **Создайте новый метод `DownloadFile`, который принимает две строки типа `string` — `remoteFile` и `localFile`.**
5. **Введите следующий код в метод `DowloadFile()`.**

```
private void DownLoadFile(string remoteFile, string localFile)
{
    // Создать объекты потока и запроса.
    FileStream localFileStream =
        new FileStream(localFile, FileMode.OpenOrCreate);
    FtpWebRequest ftpRequest =
        (FtpWebRequest)WebRequest.Create(remoteFile);

    // Настройка запроса.
    ftpRequest.Method = WebRequestMethods.Ftp.DownloadFile;
    ftpRequest.Credentials =
        new NetworkCredential("Anonymous", "");
```

```
    // Настройка ответа на запрос.
    WebResponse ftpResponse = ftpRequest.GetResponse();
    Stream ftpResponseStream = ftpResponse.GetResponseStream();
    byte[] buffer = new byte[1024];

    // Обработка запроса на загрузку данных.
    int bytesRead = ftpResponseStream.Read(buffer, 0, 1024);
    while (bytesRead > 0)
    {
        localFileStream.Write(buffer, 0, bytesRead);
        bytesRead = ftpResponseStream.Read(buffer, 0, 1024);
    }

    // Закрытие потоков.
    localFileStream.Close();
    ftpResponseStream.Close();
}
```

Код, следующий за процессом установления соединения, конфигурирует соединение и ответ на это соединение, а затем выполняет задачу. В данном случае задача состоит в том, чтобы загрузить файл с FTP-сайта. При предоставлении сетевых учетных данных для FTP-сайта вам часто нужно указывать свой адрес электронной почты в качестве пароля (второй параметр, который в примере не указан). Вы должны всегда закрывать потоки, когда выполнение задачи завершено.

6. **Вызовите метод `DownloadFile()` из обработчика события `button1_Click()` с помощью следующего кода:**

```
private void button1_Click(object sender, EventArgs e)
{
    DownLoadFile(@"ftp://ftp.yourftpsite.com/afile.txt",
                 @"c:\temp\afile.txt");
}
```


ЗАПОМНИ!

Чтобы использовать этот пример, вы должны заменить первую строку местоположением файла на своем FTP-сайте, а вторую строку — расположением на жестком диске, куда вы хотите поместить файл. В данном примере классы `WebRequest` и `WebResponse` в пространстве имен `System.Net` используются для создания более полного класса `FtpWebRequest`. Такие свойства, как `Method` загрузки и `Credentials`, упрощают вызов.

Фактически самая сложная часть этого процесса связана с объектом `FileStream`, который по-прежнему является лучшим средством для перемещения файлов и не относится к пространству имен `System.Net`. Потоки обсуждаются в главе 25, "Рыбалка в потоке", которая охватывает пространство имен `System.IO`, но они представляют ценность и для сетевых классов. Потоки

представляют собой поток данных некоторого рода, и поток информации из Интернета соответствует этим требованиям.

Это именно то, что вы делаете, когда получаете веб-страницу или файл из Интернета — получаете поток данных. Если задуматься, то это поток, иллюстрируемый строкой состояния в приложении, которая показывает процент выполнения загрузки и которая выглядит, как наполнение стакана потоком воды.

Эта концепция верна и для получения файла из WWW. Веб-протокол HTTP является еще одним протоколом, который определяет, как документ перемещается с сервера в Интернете на локальный компьютер. Код при этом выглядит поразительно похожим на код, использующий FTP, как вы можете видеть в приведенном далее фрагменте кода:

```
private void DownLoadWebFile(string remoteFile, string localFile)
{
    FileStream localFileStream =
        new FileStream(localFile, FileMode.OpenOrCreate);
    WebRequest webRequest = WebRequest.Create(remoteFile);
    webRequest.Method = WebRequestMethods.Http.Get;

    WebResponse webResponse = webRequest.GetResponse();
    Stream webResponseStream = webResponse.GetResponseStream();

    byte[] buffer = new byte[1024];
    int bytesRead = webResponseStream.Read(buffer, 0, 1024);
    while (bytesRead > 0)
    {
        localFileStream.Write(buffer, 0, bytesRead);
        bytesRead = webResponseStream.Read(buffer, 0, 1024);
    }

    localFileStream.Close();
    webResponseStream.Close();
}
```


ЗАПОМНИ!

Вы должны передать веб-адрес, так что ваш вызов подпрограммы выглядит следующим образом:

```
DownloadWebFile(@"http://your.ftp.server.com/sampleFile.bmp",
                @"c:\sampleFile.bmp");
```

В этом коде есть несколько отличий. Объект `webRequest` теперь имеет тип `WebRequest`, а не `FtpWebRequest`. Кроме того, свойство `Method` объекта `webRequest` было изменено на `WebRequestMethods.Http.Get`. Наконец свойство `Credentials` было удалено, поскольку учетные данные больше не требуются.

Отчет по электронной почте

Электронная почта является распространенным средством в сетевых системах. Если вы работаете в корпоративной среде, то имеет смысл создание более масштабного приложения, отвечающего всем требованиям к электронной почте, а не делать каждое отдельное приложение способным к работе с электронной почтой. Но если вы пишете автономный продукт, ему может потребоваться поддержка электронной почты.

Электронная почта — операция серверная, поэтому, если у вас нет почтового сервера, который вы можете использовать для отправления сообщений, ее реализация может быть сложной проблемой. Многие интернет-провайдеры из-за проблемы спама не разрешают *ретрансляцию* электронной почты, т.е. отправку исходящего сообщения без предварительной регистрации и входа в систему. В связи с этим у вас могут возникнуть проблемы с выполнением этой части рассматриваемого примера приложения.

Однако если вы находитесь в корпоративной среде, то обычно вы можете поговорить со своим администратором электронной почты и получить разрешение на использование почтового сервера. Чтобы создать функцию отправки электронной почты, выполните следующие действия.

1. **Добавьте управляющий элемент TextBox на форму в окне Design view, а затем перейдите в окно работы с кодом Code view.**
2. **Добавьте в код следующие инструкции импорта:**

```
using System.Net.Mail;
```

3. **Создайте новый метод `SendEmail`, который принимает следующие параметры типа `string` — `fromAddress`, `toAddress`, `subject` и `body`.**

   Это адреса отправителя и получателя электронной почты, тема письма и его тело.

4. **Введите в метод `SendEmail ()` следующий код.**

```
private void SendEmail(string fromAddress, string toAddress,
                       string subject, string body)
{
    // Определение сообщения
    MailMessage message =
        new MailMessage(fromAddress, toAddress, subject, body);

    // Создание соединения и отправка сообщения
    SmtpClient mailClient = new SmtpClient("localhost");
    mailClient.Send(message);

    // Освобождение сообщения и клиента
    message = null;
    mailClient = null;
}
```

Процесс начинается с создания сообщения. Затем создается клиент, который обеспечивает соединение с сервером и отправляет ему сообщение. Последний шаг заключается в освобождении сообщения и клиента, чтобы сборщик мусора мог вернуть их память системе.

Обратите внимание, что код использует в качестве имени почтового сервера `localhost`. Если у вас установлено локальное программное обеспечение сервера электронной почты, пусть даже IIS 6.0 с SMTP, этот код будет работать. В большинстве же случаев вам придется указывать иное имя почтового сервера в конструкторе `SmtpClient`. Это имя почтового сервера часто можно найти в настройках Outlook.

После того как вы напишете свой метод, его нужно вызвать после загрузки файла в обработчике событий `Button1_Click`. Измените код этой подпрограммы на следующий, вызывающий метод отправки электронной почты:

```
private void button1_Click(object sender, EventArgs e)
{
    // Загрузка файла
    DownLoadFile(@"ftp://ftp.yourftpsite.com/afile.txt",
                 @"c:\temp\afile.txt");

    // Отправка сообщения о загрузке
    SendEmail(textBox1.Text, textBox1.Text,
              "FTP Successful", "FTP Successfully downloaded");
}
```

Значение текстового поля в этом примере используется дважды: один раз — для адреса `to` и еще раз — для адреса `from`. Это не всегда необходимо, поскольку у вас может возникнуть ситуация, когда требуется, чтобы электронное письмо приходило только с адреса веб-мастера или только на ваш адрес.

ВНИМАНИЕ!

Теперь у вас должно быть достаточно кода для запуска приложения. Тем не менее пока что пример не будет работать: вы должны предоставить информацию о местоположении для FTP- и SMTP-серверов. Нажмите клавишу <F5>, чтобы запустить приложение в режиме отладки.

Когда вы щелкаете на кнопке, приложение должно загрузить файл на локальный диск, а затем отправить вам электронное письмо, сообщающее, что загрузка завершена. Тем не менее с сетевыми приложениями множество вещей может пойти не так.

» Для большинства сетевых операций компьютер, на котором запущено программное обеспечение, должен быть подключен к сети. Это не проблема для вас как разработчика, но вы должны понимать конечных пользователей, которым может потребоваться

подключение к сети для доступа к необходимым им функциям. Вы можете информировать пользователей о доступности этих функций с помощью кода получения информации о состоянии сети.

» Брандмауэры и другие сетевые устройства иногда блокируют сетевой трафик от (вполне законных) приложений. Вот некоторые примеры.

- Доступ к FTP из корпоративных сетей часто оказывается заблокированным.
- На корпоративных серверах часто блокируются функции сетевого анализа .NET. Если сервер общедоступен, это может приоткрыть лазейки для проникновения хакеров.
- Говоря о хакерах, убедитесь, что, если вы используете входящие сетевые соединения в своем приложении, то должным образом обезопасили его от вторжения.
- Особенно хрупкая в этом отношении электронная почта. Зачастую интернет-провайдеры блокируют электронную почту с адреса, который не зарегистрирован на почтовом сервере. Это означает, что если вы используете локальный сервер, ваш интернет-провайдер может заблокировать электронную почту.

» Известно, что сетевой трафик очень трудно отлаживать. Например, если приложение работает, но вы не получаете письмо от `SmtpServer`, что именно пошло не так? Вы можете об этом никогда не узнать. Похожая проблема есть и у XML-веб-сервисов — сложно обнаружить фактический код в оболочке SOAP.

Регистрация сетевой активности

Это подводит вас к следующей теме — протоколированию. Поскольку проблемы сетевой активности так трудно отлаживать и воспроизводить, Microsoft встроила некоторые инструменты для отслеживания и протоколирования сетевой активности.

Более того, как и трассировка ASP.NET, трассировка пространства имен `System.Net` полностью управляется с помощью файлов конфигурации. Чтобы иметь возможность использовать те или иные функции, вам не нужно изменять и перекомпилировать свой код. Фактически ценой очень небольшого управления файлами `config`, которые использует ваше приложение, вы даже можете предоставить отладочную информацию пользователю.

Для каждой разновидности приложения имеется файл конфигурации своего вида. Для приложений Windows Forms, которые здесь используются, файл называется `app.config` и хранится в каталоге разработки проекта. При компиляции имя файла заменяется именем приложения, и он копируется в каталог `bin` для выполнения.

Если теперь вы откроете файл app.config двойным щелчком на его записи в обозревателе решений Solution Explorer, то увидите, что он практически пуст (см. листинг 26.1). Это необычно для каркаса .NET, который ранее имел очень сложные файлы конфигурации. Мы добавим кое-что в этот файл, чтобы включить трассировку.

Листинг 26.1. Файл app.config по умолчанию

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0"
                          sku=".NETFramework,Version=v4.5.2"/>
    </startup>
</configuration>
```

Сначала нужно добавить новый источник для пространства имен System.Net. Затем — переключатель в раздел Switches для добавленного источника. Наконец следует добавить в этот раздел SharedListener и установить файл для автоматического сброса трассируемой информации. Готовый файл app.config с выделенными полужирным шрифтом добавлениями показан в листинге 26.2.

Листинг 26.2. Окончательный вид файла app.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0"
                          sku=".NETFramework,Version=v4.5.2"/>
    </startup>
    <system.diagnostics>
        <sources>
            <source name="System.Net">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
        </sources>
        <switches>
            <add name="System.Net" value="Verbose" />
        </switches>
        <sharedListeners>
            <add name="System.Net"
                 type="System.Diagnostics.TextWriterTraceListener"
                 initializeData="my.log"/>
        </sharedListeners>
        <trace autoflush="true" />
    </system.diagnostics>
</configuration>
```

Запустите приложение еще раз и взгляните на окно вывода Output. Из-за внесенных в файл конфигурации изменений в окне отображается расширенная информация; кроме того, записывается журнальный файл. В среде разработки он находится в каталоге `bin/debug` вашего проекта. Возможно, чтобы увидеть этот файл, вам придется щелкнуть на кнопке Show All Files в верхней части Solution Explorer.

В этой папке вы должны увидеть файл с именем `my.log`, куда `SharedListener`, который вы добавили в файл `app.config`, направил протоколируемую информацию. В листинге 26.3 показано, как может выглядеть содержимое этого файла. Конкретные URL, номера ссылок и прочие значения объектов, конечно же, в ваших выходных данных будут иными.

Листинг 26.3. Журнальная информация

```
System.Net Information: 0 : WebRequest::Create(ftp://ftp.
    csharpfordummies.net/sample.bmp)
System.Net Information: 0 : Exiting WebRequest::Create() ->
    FtpWebRequest#37460558
System.Net Information: 0 : FtpWebRequest#37460558::GetResponse()
System.Net Information: 0 : Exiting FtpWebRequest#37460558
    ::GetResponse()
System.Net Information: 0 : Associating Message#59487907 with
    HeaderCollection#23085090
System.Net Information: 0 : HeaderCollection#23085090
    ::Set(mime-version=1.0)
System.Net Information: 0 : Associating MailMessage#6964596
    with Message#59487907
System.Net Information: 0 : SmtpClient::.ctor(host=24.123.157.3)
System.Net Information: 0 : Associating SmtpClient#17113003 with
    SmtpTransport#30544512
System.Net Information: 0 : Exiting SmtpClient::.ctor() ->
    SmtpClient#17113003
System.Net Information: 0 : SmtpClient#17113003
    ::Send(MailMessage#6964596)
System.Net Information: 0 : SmtpClient#17113003
    ::Send(DeliveryMethod=Network)
System.Net Information: 0 : Associating SmtpClient#17113003 with
    MailMessage#6964596
System.Net Information: 0 : Associating SmtpTransport#30544512 with
    SmtpConnection#44365459
System.Net Information: 0 : Associating SmtpConnection#44365459 with
    ServicePoint#7044526
System.Net Information: 0 : Associating SmtpConnection#44365459 with
    SmtpPooledStream#20390146
System.Net Information: 0 : HeaderCollection#30689639
    ::Set(content-transferencoding=base64)
System.Net Information: 0 : HeaderCollection#30689639
    ::Set(content-transferencoding=quoted-printable)
System.Net Information: 0 : HeaderCollection#23085090
```

```
    ::Remove(x-receiver)
System.Net Information: 0 : HeaderCollection#23085090
    ::Set(from=bill@sempf.net)
System.Net Information: 0 : HeaderCollection#23085090
    ::Set(to=bill@sempf.net)
System.Net Information: 0 : HeaderCollection#23085090
    ::Set(date=1 Apr 2010 16:32:32 -0500)
System.Net Information: 0 : HeaderCollection#23085090
    ::Set(subject=FTPSuccessful)
System.Net Information: 0 : HeaderCollection#23085090
    ::Get(mime-version)
System.Net Information: 0 : HeaderCollection#23085090::Get(from)
System.Net Information: 0 : HeaderCollection#23085090::Get(to)
System.Net Information: 0 : HeaderCollection#23085090::Get(date)
System.Net Information: 0 : HeaderCollection#23085090::Get(subject)
System.Net Information: 0 : HeaderCollection#30689639
    ::Get(content-type)
System.Net Information: 0 : HeaderCollection#30689639
    ::Get(content-transferencoding)
System.Net Information: 0 : Exiting SmtpClient#17113003::Send()
```

Просмотрев файл, вы увидите, что приведенная в нем информация значительно упрощает отладку. Кроме того, поскольку все записи следуют в порядке выполнения действий, гораздо проще выяснить, где именно произошла ошибка.

Глава 27

Создание изображений

В ЭТОЙ ГЛАВЕ...

- **Экскурсия по пространству имен** `System.Drawing`
- **Классы для работы с изображениями**
- **Простейшая игра с использованием** `System.Drawing`

Никто не собирается писать очередную версию игры Bioshock с использованием C#. Это не тот язык, который используется для приложений с интенсивной графикой типа "стрелялок".

Тем не менее C# обладает достаточными графическими возможностями, сосредоточенными в классах `System.Drawing`. Хотя для некоторых областей эти классы слишком примитивны и их применение может привести к тому, что вам придется писать больше кода, чем необходимо, есть мало задач, с которыми эти классы не смогли бы справиться.

Графические возможности, предоставляемые .NET Framework, разделяются на четыре логические области с соответствующими пространствами имен, предоставленными Microsoft. Все общие графические возможности находятся в пространстве имен `System.Drawing`. Есть также некоторые специализированные пространства имен.

- `System.Drawing.2D` обладает расширенными функциональными возможностями векторного черчения.
- `System.Drawing.Imaging` посвящено графическим растровым форматам, таким как файлы `.bmp` и `.jpg`.
- `System.Drawing.Text` обладает расширенными функциональными возможностями вывода текста.

Эта глава посвящена базовому пространству имен и охватывает только основы работы с графикой в C# (подробное обсуждение каждого аспекта графики может легко занять целую книгу).

Знакомство с System.Drawing

Даже на самом высоком уровне графическое программирование состоит из рисования многоугольников, заполнения их цветом и маркировки их текстом — все на каком-то холсте. Неудивительно, что это оставляет вас с четырьмя объектами, которые составляют ядро графического кода: графикой, перьями, кистями и текстом.

Графика

Вообще говоря, класс `Graphics` создает объект, который является вашей палитрой. Это холст. Все методы и свойства объекта `Graphics` предназначены для создания подходящей для ваших нужд области, в которой вы рисуете.

Кроме того, большинство графических методов других классов платформы предоставляют в качестве вывода объект `Graphics`. Например, можно вызвать метод `System.Web.Forms.Control.CreateGraphics` из приложения Windows Forms и получить объект `Graphics`, который позволяет рисовать на соответствующем управляющем элементе формы вашего проекта. Можно также обработать событие `Paint` формы и проверить его свойство `Graphics`.

Графические объекты для рисования и заливки используют перья и кисти (обсуждаемые далее в этой главе в соответствующих разделах). Графические объекты имеют такие методы.

- `DrawRectangle`
- `FillRectangle`
- `DrawCircle`
- `FillCircle`
- `DrawBezier`
- `DrawLine`

Эти методы принимают в качестве параметров перья и кисти. Вы можете подумать "Чем мне может помочь черчение окружности?" Но вспомните, что даже самые сложные графические объекты состоят из примитивных кругов и прямоугольников — из тысяч таких фигур. Вся хитрость заключается в использовании математики, позволяющей собрать в единое целое множество кругов и квадратов, пока не получится полное изображение.

Перья

Вы используете для рисования линий и кривых перья (pen). Сложная графика состоит из многоугольников, и эти многоугольники состоят из линий, и эти линии генерируются перьями. Перья имеют следующие свойства.

- `Color`
- `DashStyle`
- `EndCap`
- `Width`

Идея ясна: перья используются, чтобы рисовать разные вещи. Эти свойства используются перьями для определения, что и как должно быть нарисовано.

Кисти

Кисти предназначены для рисования внутренностей многоугольников. Хотя для рисования фигур и используются перья, но чтобы заполнить фигуры градиентами, узорами или цветами, используются кисти. Кисти обычно передаются в качестве параметров в методы `Draw...` независимо от перьев. Когда перо рисует форму, кисть используется, чтобы заполнить ее, так же как вы делали это в детском саду с карандашами и раскрасками. (Правда, нарисованное кистью всегда остается внутри линий.)

Однако не ищите класс `Brush`. Это место для хранения настоящих кистей со странными названиями. Кисти могут быть настроены пользователями "по индивидуальному заказу", но очень многое можно сделать с помощью предопределенных кистей, поставляемых с каркасом. Вот некоторые из них.

- `SolidBrush`
- `TextureBrush`
- `HatchBrush`
- `PathGradientBrush`

Перья используются для передачи в методы `Draw...` объекта `Graphics`, кисти же передаются в методы `Fill...`, которые изображают многоугольники.

Текст

Текст выводится с помощью комбинации шрифтов и кистей. Так же, как и перья, класс `Font` использует кисти для заполнения текстовых строк.

`System.Drawing.Text` содержит коллекции всех шрифтов, установленных в системе, в которой запущена ваша программа, или инсталлированных как часть вашего приложения. `System.Drawing.Font` имеет ряд типографских свойств.

- `Bold`
- `Size`
- `Style`
- `Underline`

ПЕЧАТЬ ФОРМЫ

В VB6 и более ранних версиях одним из наиболее распространенных способов получения информации на бумаге была печать формы. Данная функциональность была утрачена в .NET, но вернулась в Power Pack и теперь встроена в Visual Studio 2008 и выше. Она доступна во всех языках, но программисты VB используют ее чаще всех.

Если вам нужно создать отчет, вы должны использовать Microsoft Report Viewer, который не описан в этой книге. Если вы просто хотите вывести текст и изображения на пользовательский принтер, вам должен помочь компонент `PrintForm`.

Чтобы использовать компонент `PrintForm`, перетащите его из панели инструментов в форму в представлении Design View, и он появится на панели компонентов. В обработчике события установите свойство `Form` компонента, а затем вызовите команду печати:

```
using PrintForm printForm = new PrintForm
    .Form =TheFormIWantPrinted
    .PrintAction = PrintToPrinter
    .Print()
end using
```

Классы рисования и каркас .NET

Пространство имен `System.Drawing` разбивает процесс рисования на два этапа.

1. **Создание объекта `System.Drawing.Graphics`.**
2. **Применение инструментов пространства имен `System.Drawing` для рисования на нем.**

Это кажется простым делом, и это так и есть. Первый шаг состоит в получении объекта `Graphics`. Эти объекты получаются в основном из существующих изображений и из Windows Forms.

Чтобы получить объект `Graphics` из существующего изображения, рассмотрите объект `Bitmap`. Объект `Bitmap` — это отличный инструмент, который позволяет вам создать объект, используя существующий файл изображения. Он создает палитру, основанную на растровом изображении (например, из файла JPEG), которое уже находится на вашем жестком диске. Это удобный инструмент, особенно для веб-изображений.

```
Bitmap currentBitmap = new Bitmap(@"c:\images\myImage.jpg");
Graphics palette = Graphics.FromImage(currentBitmap);
```

Теперь объект `myPalette` представляет собой объект типа `Graphics`, высота и ширина которого основаны на изображении в `myBitmap`. Более того, основа изображения `myPalette` выглядит точно так же, как изображение, на которое ссылается объект `myBitmap`.

Чтобы рисовать непосредственно на этом изображении, как если бы это был пустой холст, можно использовать перья, кисти и шрифты в классе `Graphics`. Можно, например, использовать шрифт для размещения текста на изображении перед его отображением на веб-странице, а также использовать другие элементы `Graphics` для изменения формата изображения “на лету”.

Другой способ получения объекта `Graphics` — из Windows Forms. Для этого нужен метод `System.Windows.Forms.Control.CreateGraphics`, который дает новую палитру, основанную на поверхности рисования элемента управления, на который он ссылается. Если это форма, он наследует высоту и ширину формы и имеет цвет ее фона. Вы можете использовать перья и кисти, чтобы рисовать прямо на форме.

Помните, что даже самая сложная трехмерная графика — это просто раскрашенные многоугольники, и вы можете создавать их с помощью класса `System.Drawing`.

Использование пространства имен System.Drawing

Многим нравятся игры, и одной из распространенных игр является карточная игра криббедж[1]. Представим, что вы в отпуске и хотите немного поиграть в нее. У вас есть карты, не хватает доски для криббеджа.

Но если у вас есть ноутбук, Visual Studio и пространство имен `System.Drawing`, то за несколько часов вы можете создать приложение, которое будет работать в качестве необходимой доски. Пример, приведенный в следующих разделах, не является достаточно полным (для этого потребовалось бы куда больше кода), но включает в себя достаточное количество кода, чтобы начать работу, и достаточно функционален, чтобы позволить вам играть в игру.

Приступая к работе

Криббедж — это карточная игра, в которой карты на руках игроков пересчитываются в очки, и выигрывает первый игрок, набравший 121 очко. Подсчет очков ведется с помощью специальной доски, которая состоит из двух линий отверстий для колышков, обычно по 120 отверстий (иногда используются 60 отверстий и два прохода по ним). Типичная доска для криббеджа показана на рис. 27.1. Доски для криббеджа бывают разных стилей (если вам интересно, посетите сайт `www.cribbage.org`; на нем имеется галерея почти из 100 досок, от простых до самых причудливых).

В этом примере вы создадите изображение игрового поля для приложения, которое ведет счет в игре криббедж, но мы не будем заставлять C# еще и играть в карты вместо вас. Доска в рассматриваемом приложении имеет по 40 лунок на каждой из трех пар линий — стандартная доска для двух игроков, играющих до 120 очков, показанная на рис. 27.2. Первая задача — нарисовать игровое поле, а затем рисовать колышки по мере изменения очков игроков, вводимых в текстовые поля.

Суть заключается в следующем: игроки играют "вручную" и вводят итоговые результаты в текстовые поля под своими именами (см. рис. 27.2). После ввода итоговый счет рядом с именем игрока обновляется, а колышек перемещается на доске. В следующий раз при вводе данных игроком колышек перемещается вперед, задний колышек перемещается на его место. Изобретатель криббеджа был параноиком, и задний колышек делает обман менее вероятным.

[1] Не знакомые с игрой могут узнать о ней в Википедии: `https://ru.wikipedia.org/wiki/Криббедж`. — *Примеч. пер.*

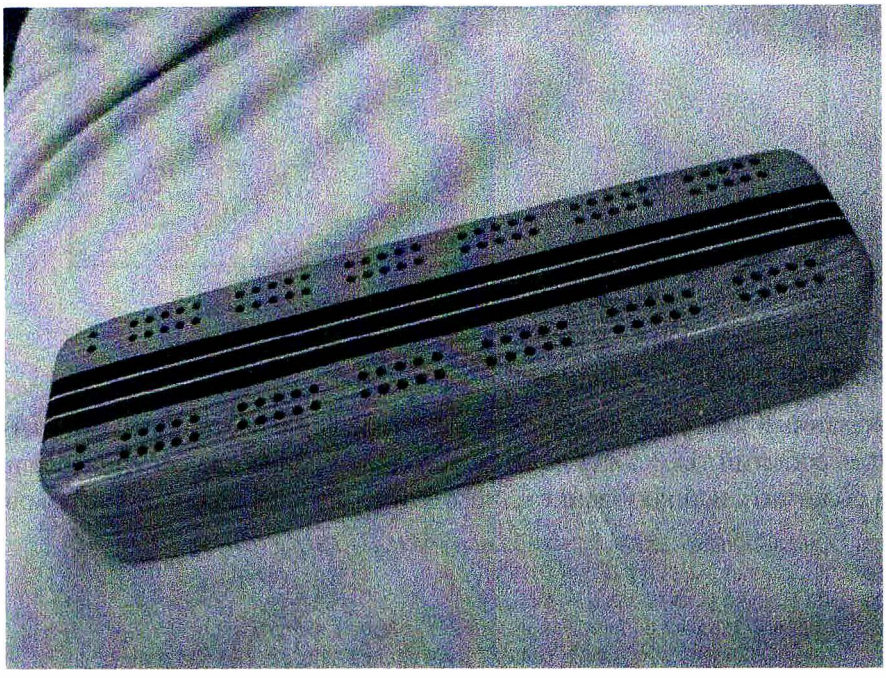


Рис. 27.1. Традиционная доска для криббеджа

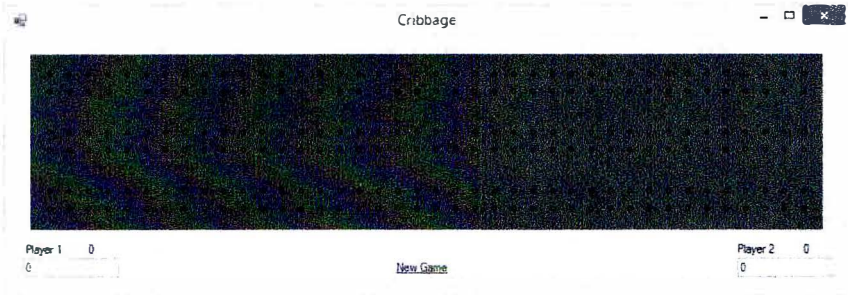


Рис. 27.2. Цифровая доска для криббеджа

Настройка проекта

Для начала создадим игровую поверхность. Мы настраиваем игровую доску, показанную на рис. 27.2, не рисуя саму доску — позже вы узнаете, как нарисовать ее с помощью объектов `System.Drawing`. Когда вы будете готовы начать создавать бизнес-правила, окно программы должно выглядеть так, как показано на рис. 27.3. Управляющие элементы окна (перечисляемые слева направо) называются `Player1Points` (`Label`), `Player1` (`TextBox`), `WinMessage` (`Label`), `StartGame` (`LinkLabel`), `Player2Points` (`Label`) и `Player2` (`TextBox`).

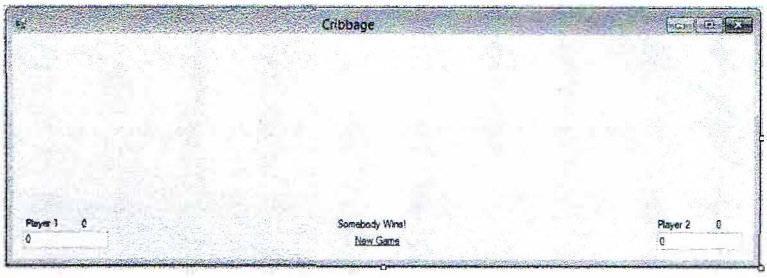


Рис. 27.3. Основное окно программы

Обработка счета

Показанный далее метод обрабатывает изменения счета при вызове из обработчиков событий `TextChanged`. Этот метод намеренно сделан универсальным, чтобы облегчить использование одного и того же кода для обоих игроков.

```
// Поля, используемые для отслеживания счета.
private int Player1LastTotal = 0;
private int Player2LastTotal = 0;
private void HandleScore(TextBox scoreBox, Label points,
                         Label otherPlayer, ref Int32 lastScore)
{
    try
    {
        if (0 > Int32.Parse(scoreBox.Text) |
                Int32.Parse(scoreBox.Text) > 27)
        {
            // Вывод сообщения об ошибке и передача фокуса
            WinMessage.Text = "Score must be between 0 and 27";
            scoreBox.Focus();
        }
        else
        {
            // Очистка сообщения об ошибке.
            WinMessage.Text = "";
            // Обновление последнего счета.
            lastScore = Int32.Parse(points.Text);
            //Add the score written to the points
            points.Text = (Int32.Parse(points.Text) +
                           Int32.Parse(scoreBox.Text)).ToString();
        }
    }
    catch (System.InvalidCastException ext)
    {
        // Нечто, не являющееся числом
        if (scoreBox.Text.Length > 0)
        {
            WinMessage.Text = "Score must be a number";
        }
    }
```

```
    catch (Exception ex)
    {
        //Eek!
        MessageBox.Show("Something went wrong! " + ex.Message);
    }

    // Проверка счета
    if (Int32.Parse(points.Text) > 120)
    {
        if (Int32.Parse(points.Text) /
                Int32.Parse(otherPlayer.Text) > 1.5)
        {
            WinMessage.Text = scoreBox.Name.Substring(0,
                              scoreBox.Name.Length - 6) +
                              " Skunked 'em!!!";
        }
        else
        {
            WinMessage.Text = scoreBox.Name.Substring(0,
                              scoreBox.Name.Length - 6) +
                              " Won!!";
        }

        WinMessage.Visible = true;
    }
}
```

Создание подключения к событию

Конечно, если у вас есть события, должны быть и их обработчики. По очереди дважды щелкните на `Player1` и `Player2`, чтобы создать следующие обработчики событий:

```
private void Player1_TextChanged(object sender, EventArgs e)
{
    // Обработка счета
    HandleScore(Player1, Player1Points, Player2Points,
                ref Player1LastTotal);
    // Обновление доски
    Form1.ActiveForm.Invalidate();
}
private void Player2_TextChanged(object sender, EventArgs e)
{
    // Обработка счета
    HandleScore(Player2, Player2Points, Player1Points,
                ref Player2LastTotal);
    // Обновление доски
    Form1.ActiveForm.Invalidate();
}
```

Обратите внимание, что вы должны передавать закрытое поле, используемое для хранения счета предыдущего игрока, по ссылке. В противном случае поля не будут обновлены.

Кроме того, вы должны вызвать `Form1.ActiveForm.Invalidate()`. В противном случае доска не будет перерисована, а значит, вы не увидите перемещение колышков.

Рисование доски

Чтобы создать изображение доски, приложение должно выводить его прямо на форме. Это означает получение доступа к объекту `Graphics` через объект `PaintEventArgs`, передаваемый приложению во время каждого события перерисовки. Вам нужно выполнить следующие задачи.

- Изобразить коричневую доску с помощью кисти.
- Нарисовать шесть строк маленьких кружков с помощью пера.
- Заполнить нужный кружок при корректном счете.
- Выполнить очистку.

Показанный далее метод перерисовывает доску при каждом вызове. Чтобы сделать предназначение метода более понятным, он назван `CribbageBoard_Paint()`.

```
private void CribbageBoard_Paint(object sender, PaintEventArgs e)
{
    // Получение объекта Graphics.
    Graphics g = e.Graphics;
    // Создание доски
    SolidBrush brownBrush = new SolidBrush(Color.Brown);
    g.FillRectangle(brownBrush, new Rectangle(20, 20, 820, 180));

    // Рисование 240 маленьких дырочек - три линии 40x2
    int rows = 0;
    int columns = 0;
    int scoreBeingDrawn = 0;
    Pen blackPen = new Pen(System.Drawing.Color.Black, 1);
    SolidBrush blackBrush = new SolidBrush(Color.Black);
    SolidBrush redBrush = new SolidBrush(Color.Red);

    // Вывод 6 строк
    for (rows = 40; rows <= 160; rows += 60)
    {
        // По 40 столбцов через 20 точек
        for (columns = 40; columns <= 820; columns += 20)
        {
            // Вычисление выводимого счета
            scoreBeingDrawn = ((columns - 20) / 20) +
                              ((((rows + 20) / 60) - 1) * 40);
```

```
            // Вывод Player1
            if (scoreBeingDrawn == Int32.Parse(Player1Points.Text))
            {
                g.FillEllipse(blackBrush, columns-2, rows-2, 6, 6);
            }
            else if (scoreBeingDrawn == Player1LastTotal)
            {
                g.FillEllipse(redBrush, columns-2, rows-2, 6, 6);
            }
            else
            {
                g.DrawEllipse(blackPen, columns-2, rows-2, 4, 4);
            }

            // Вывод Player2
            if (scoreBeingDrawn == Int32.Parse(Player2Points.Text))
            {
                g.FillEllipse(blackBrush, columns-2, rows+16, 6, 6);
            }
            else if (scoreBeingDrawn == Player2LastTotal)
            {
                g.FillEllipse(redBrush, columns-2, rows+16, 6, 6);
            }
            else
            {
                g.DrawEllipse(blackPen, columns-2, rows+16, 4, 4);
            }
        }
    }

    // Выполнение очистки.
    g.Dispose();
    brownBrush.Dispose();
    blackPen.Dispose();
}
```

ЗАПОМНИ!

Для правильной работы обработчика событий `CribbageBoard_Paint()` необходимо связать его с формой. В представлении Design View выберите `Form1`. Щелкните на кнопке Events в верхней части окна Properties, чтобы отобразить список событий, связанных с `Form1`. Щелкните на раскрывающемся списке для события `Paint` и выберите в нем `CribbageBoard_Paint`.

Запуск новой игры

Последнее, что нужно сделать, — это создать метод запуска новой игры, в котором в игру вступают `LinkLabel` и `StartGame`. Вот код, выполняющий настройки для новой игры:

```
private void StartGame_LinkClicked(object sender,
                                   LinkLabelLinkClickedEventArgs e)
{
```

```
    // Установка нулевого счета.
    Player1.Text = "0";
    Player2.Text = "0";

    Player1Points.Text = "0";
    Player2Points.Text = "0";

    Player1LastTotal = 0;
    Player2LastTotal = 0;

    // Сброс текста.
    WinMessage.Text = "";
}
```

Трудно поверить, но именно так пишутся и крупномасштабные игры. Конечно, в больших графических играх куда больше принятий решений `if-then`, но идея остается той же.

Кроме того, в больших играх иногда используются растровые изображения, а не активное рисование. Например, в нашем приложении для крибббеджа можно использовать растровое изображение колышка, а не просто заполнять эллипс черной или красной кистью.

Предметный указатель

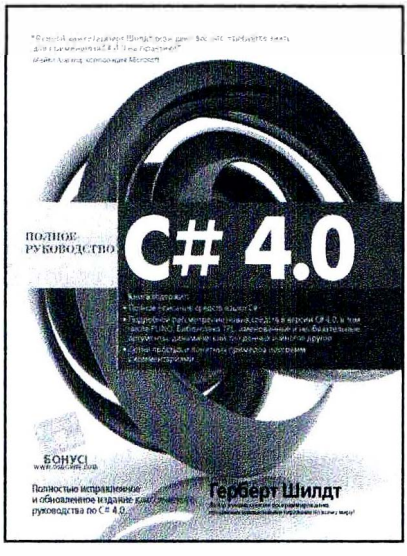
Н

О

П

C# 4.0
ПОЛНОЕ РУКОВОДСТВО

Герберт Шилдт



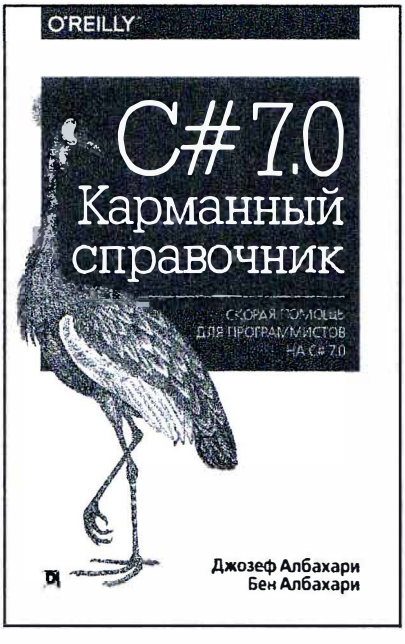
www.williamspublishing.com

В этом полном руководстве по C# 4.0 — языку программирования, разработанному специально для среды .NET, — детально рассмотрены все основные средства языка: типы данных, операторы, управляющие операторы, классы, интерфейсы, методы, делегаты, индексаторы, события, указатели, обобщения, коллекции, основные библиотеки классов, средства многопоточного программирования и директивы препроцессора. Подробно описаны новые возможности C#, в том числе PLINQ, библиотека TPL, динамический тип данных, а также именованные и необязательные аргументы. Это справочное пособие снабжено массой полезных советов авторитетного автора и сотнями примеров программ с комментариями, благодаря которым они становятся понятными любому читателю независимо от уровня его подготовки. Книга рассчитана на широкий круг читателей, интересующихся программированием на C#.

ISBN 978-5-907114-49-4 **в продаже**

C# 7.0
КАРМАННЫЙ СПРАВОЧНИК

Джозеф Албахари
Бен Албахари



www.williamspublishing.com

Когда вам нужны ответы на вопросы по программированию на языке C# 7.0, этот узкоспециализированный справочник предложит именно то, что необходимо знать — безо всяких длинных введений или раздутых примеров. Легкое в чтении и идеальное в качестве краткого справочника, данное руководство поможет опытным программистам на C#, Java и C++ быстро ознакомиться с последней версией языка C#.

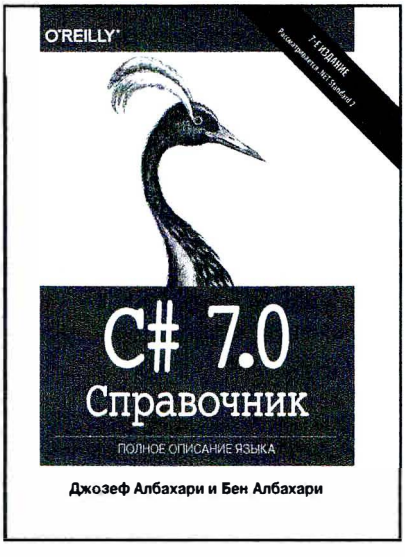
Эта книга написана авторами книги *C# 7.0. Справочник. Полное описание языка* и раскрывает все особенности языка C# 7.0.

- Фундаментальные основы C#
- Новые средства C# 7.0, включая кортежи, сопоставление по шаблону и деконструкторы
- Более сложные темы: перегрузка операций, ограничения типов, итераторы, типы, допускающие `null`, подъем операций, лямбда-выражения и замыкания
- Язык LINQ: последовательности, отложенное выполнение, стандартные операции запросов и выражения запросов
- Небезопасный код и указатели, специальные атрибуты, директивы препроцессора и XML-документация

ISBN 978-5-9909446-1-9 **в продаже**

C# 7.0
СПРАВОЧНИК
ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА
7-Е ИЗДАНИЕ

Джозеф Албахари
Бен Албахарин



www.dialektika.com

Когда у вас возникают вопросы по языку C# 7.0 или среде CLR и основным сборкам .NET Framework, это ставшее бестселлером руководство предложит все необходимые ответы. С момента представления в 2000 году C# стал языком с замечательной гибкостью и широким размахом, но такое непрекращающееся развитие означает, что по-прежнему есть многие вещи, которые предстоит изучить. Организованное вокруг концепций и сценариев использования, основательно обновленное седьмое издание книги снабдит программистов средней и высокой квалификации лаконичным планом получения знаний по C# и .NET. Погрузитесь в него и выясните, почему данное руководство считается исчерпывающим справочником по языку C#.

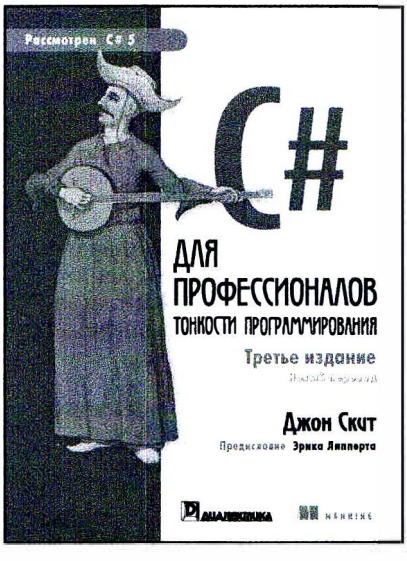
- Освойте должным образом все аспекты языка C#, от основ синтаксиса и переменных до таких сложных тем, как указатели и перегрузка операций
- Тщательно исследуйте LINQ с помощью трех глав, специально посвященных этой теме
- Узнайте о динамическом, асинхронном и параллельном программировании

ISBN 978-5-6040043-7-1

в продаже

C#
ПРОГРАММИРОВАНИЕ ДЛЯ ПРОФЕССИОНАЛОВ
3-Е ИЗДАНИЕ

Джон Скит



www.dialektika.com

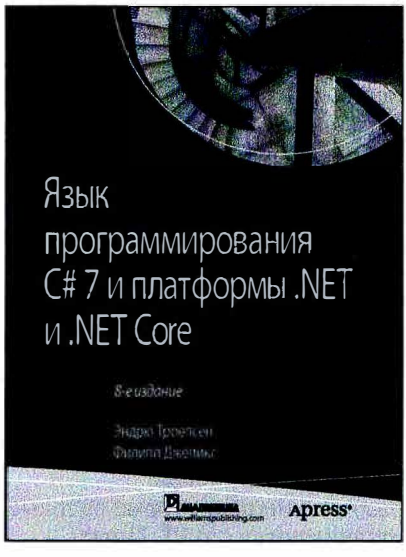
Если вы занимаетесь разработкой приложений .NET, то будете использовать C# как при построении сложного приложения уровня предприятия, так и при ускоренном написании какого-нибудь чернового приложения. В C# 5 можно делать удивительные вещи с помощью обобщений, лямбда-выражений, динамической типизации, LINQ, итераторных блоков и других средств. Однако прежде их необходимо должным образом изучить. Это издание было полностью пересмотрено с целью раскрытия новых средств версии C# 5, включая тонкости написания сопровождаемого асинхронного кода. Вы увидите всю мощь языка C# в действии и научитесь работать с ценнейшими средствами, которые эффективно впишутся в применяемый набор инструментов. Кроме того, вы узнаете, как избегать скрытых ловушек при программировании на C# с помощью простых и понятных объяснений вопросов, касающихся внутреннего устройства языка.

ISBN 978-5-907114-62-3

в продаже

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 7 И ПЛАТФОРМЫ .NET И .NET CORE
8-Е ИЗДАНИЕ

Эндрю Троелсен
Филипп Джепикс



www.williamspublishing.com

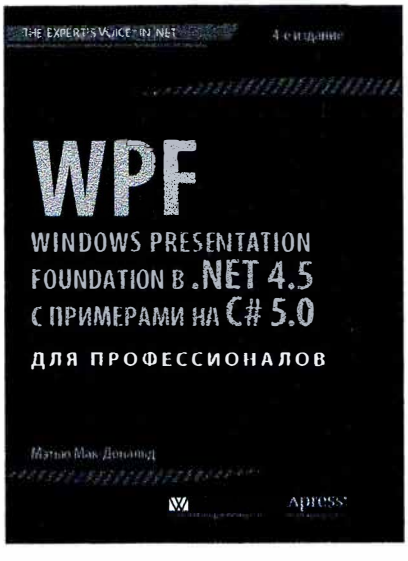
Эта классическая книга представляет собой всеобъемлющий источник сведений о языке программирования C# и о связанной с ним инфраструктуре. В 8-м издании книги вы найдете описание функциональных возможностей самых последних версий C# 7.0 и 7.1 и .NET 4.7, а также совершенно новые главы о легковесной межплатформенной инфраструктуре Microsoft .NET Core, включая версию .NET Core 2.0. Книга охватывает ASP.NET Core, Entity Framework (EF) Core и т.д. наряду с последними обновлениями платформы .NET, в том числе внесенными в Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF) и ASP.NET MVC. Книга предназначена для опытных разработчиков ПО, заинтересованных в освоении новых средств .NET 4.7, .NET Core и языка C#. Она будет служить всеобъемлющим руководством и настольным справочником как для тех, кто впервые переходит на платформу .NET, так и для тех, кто ранее писал приложения для предшествующих версий .NET.

ISBN 978-5-6040723-1-8

в продаже

WPF: WINDOWS PRESENTATION FOUNDATION В .NET 4.5 С ПРИМЕРАМИ НА C# 5.0
ДЛЯ ПРОФЕССИОНАЛОВ

МЭТЬЮ МАК-ДОНАЛЬД



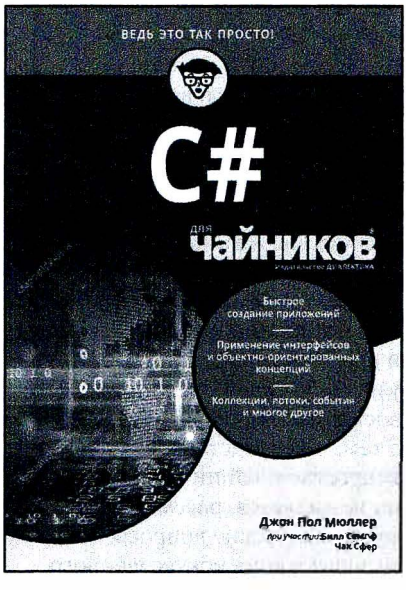
www.williamspublishing.com

Эта книга представляет собой исчерпывающее авторитетное руководство по внутренней работе WPF. Благодаря серьезным примерам и практическим рекомендациям, вы изучите все, что необходимо знать для профессионального использования WPF. Книга начинается с построения прочного фундамента из элементарных концепций, подкрепленного существующими знаниями языка C#. Затем предлагается обсуждение сложных концепций с их демонстрацией на полезных примерах, которые подчеркивают получаемую экономию времени и затраченных усилий. Книга рассчитана на разработчиков, которые впервые сталкиваются с WPF. Опыт программирования на C# и знание базовой архитектуры .NET поможет быстрее разобраться с примерами, но все необходимые концепции кратко объясняются с самого начала.

ISBN 978-5-8459-1854-3 в продаже

C# ДЛЯ ЧАЙНИКОВ

Джон Пол Мюллер



www.dialektika.com

Даже если вы никогда не имели дела с программированием, эта книга поможет вам освоить язык C# и научиться писать на нем программы любой сложности. Для читателей, которые уже знакомы с каким-либо языком программирования, процесс изучения C# только упростится, но иметь опыт программирования для чтения книги совершенно необязательно.

Из этой книги вы узнаете не только о типах, конструкциях и операторах языка C#, но и о ключевых концепциях объектно-ориентированного программирования, реализованных в этом языке, который в настоящее время представляет собой один из наиболее приспособленных для создания программ для Windows инструментов.

Если вы в начале большого пути в программирование, смелее покупайте эту книгу: она послужит вам отличным путеводителем, который облегчит ваши первые шаги на этом длинном, но очень увлекательном пути.

ISBN 978-5-907144-43-9

в продаже

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ ДЛЯ ЧАЙНИКОВ

Джон Пол Мюллер
Лука Массарон

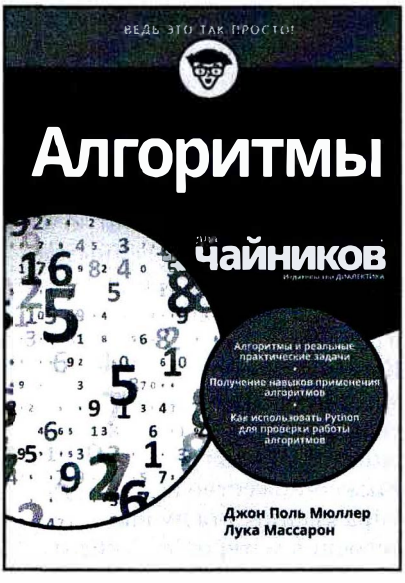
www.dialektika.com

Искусственный интеллект является захватывающим и немного жутковатым. Он вокруг нас. Искусственный интеллект помогает защитить мошенничества, контролировать расписание медицинских процедур, он способен работать в клиентской службе и даже помогает вам в выборе телешоу и приборке вашего дома. Хотите узнать больше? Эта книга восполняет пробелы, знакомя вас с тем, что представляет собой искусственный интеллект и чем он не является, рассматриваются также этические вопросы использования искусственного интеллекта, его современное применение и некоторые из удивительных вещей, на которые он, вероятно, будет способен завтра. Будь вы технофилом или просто любопытны, вы будете очарованы тем, что узнаете!

ISBN 978-5-907114-57-9 **в продаже**

АЛГОРИТМЫ ДЛЯ ЧАЙНИКОВ

Джон Поль Мюллер
Лука Массарон



www.dialektika.com

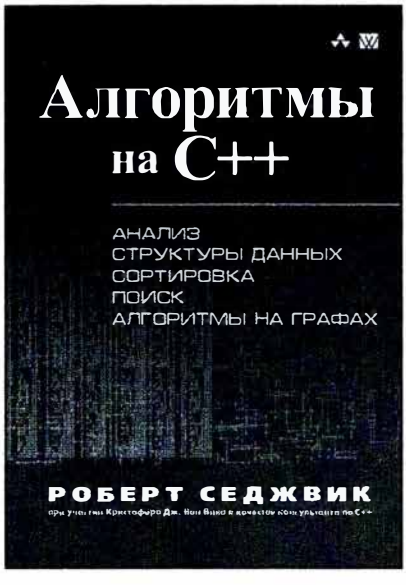
Эта книга — действительно книга для чайников, поскольку основная ее задача не научить программировать реализации тех или иных давно известных алгоритмов, а познакомить вас с тем, что же такое алгоритмы, как они влияют на нашу повседневную жизнь, и каково состояние дел в этой области человеческих знаний сегодня.
В книге рассматривается крайне широкий спектр вопросов, связанных с алгоритмами — это и стандартные сортировка и поиск, и работа с графами (но с уклоном не в стандартные базовые алгоритмы, а в приложении их к таким явлениям сегодняшнего дня, как, например, социальные сети), работа с большими данными и вопросы искусственного интеллекта.
При этом материал книги — не просто отвлеченный рассказ о том или ином аспекте современных алгоритмов, но и демонстрация реализаций алгоритмов с конкретными примерами на языке программирования Python.
Книга будет полезна всем, кто интересуется современным состоянием дел в области программирования и алгоритмов.

ISBN 978-5-9909446-2-6

в продаже

АЛГОРИТМЫ НА C++
АНАЛИЗ, СТРУКТУРЫ ДАННЫХ, СОРТИРОВКА, ПОИСК, АЛГОРИТМЫ НА ГРАФАХ

Роберт Седжвик



www.williamspublishing.com

Эта классическая книга удачно сочетает в себе теорию и практику, что делает ее популярной у программистов на протяжении многих лет. Кристофер Ван Вик и Седжвик разработали новые лаконичные реализации на C++, которые естественным и наглядным образом описывают методы и могут применяться в реальных приложениях. Каждая часть содержит новые алгоритмы и реализации, усовершенствованные описания и диаграммы, а также множество новых упражнений для лучшего усвоения материала. Акцент на АТД расширяет диапазон применения программ и лучше соотносится с современными средами объектно-ориентированного программирования. Книга предназначена для широкого круга разработчиков и студентов.

ISBN 978-5-8459-2070-6

в продаже

...FOR Dummies ™
BESTSELLING BOOK SERIES

C# для чайников®

...ДЛЯ чайников ™
СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

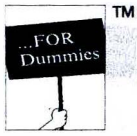
Шпаргалка

Операторы

Приоритет	Операторы	Унарность	Ассоциативность
Высокий	`() [] . new typeof`	Унарный	Слева направо
	`! ~ + - ++ -- (приведение типа)`	Унарный	Слева направо
	`* / %`	Бинарный	Слева направо
	`+ -`	Бинарный	Слева направо
	`< <= > >= is as`	Бинарный	Слева направо
	`== !=`	Бинарный	Слева направо
	`&`	Бинарный	Слева направо
	`^`	Бинарный	Слева направо
	`|`	Бинарный	Слева направо
	`&&`	Бинарный	Слева направо
	`||`	Бинарный	Слева направо
	`?:`	Тернарный	Справа налево
Низкий	`= *= /= %= += -= &= ^= |= <<= >>=`	Бинарный	Справа налево

Типы целочисленных переменных

Тип	Размер, байты	Диапазон значений	Использование
byte	1	От 0 до 255	`byte b = 12;`
short	2	От -32,768 до 32,767	`short sn = -123;`
ushort	2	От 0 до 65,535	`ushort usn = 123;`
int	4	От -2,147,483,648 до 2,147,483,647	`int n = 123;`
uint	4	От 0 до 4,294,967,295	`uint un = 123U;`
long	8	От -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807	`long l = 123L;`
ulong	8	От 0 до 18,446,744,073,709,551,615	`long ul = 123UL;`



BESTSELLING
BOOK
SERIES

C# для чайников®

СЕРИЯ
КОМПЬЮТЕРНЫХ
КНИГ ОТ
ДИАЛЕКТИКИ

Шпаргалка

Типы переменных с плавающей точкой

Тип	Размер, байты	Диапазон	Точность, цифры	Использование
float	8	От 1.5*10-45 до 3.4*1038	6–7	float f = 1.2F;
double	16	От 5.0*10-324 до 1.7*10308	15–16	double d = 1.2;

Другие типы переменных

Тип	Диапазон	Использование
decimal	До 28 цифр	decimal d = 123M;
BigInteger	—	Слишком сложно для шпаргалки
char	От 0 до 65,535 (коды в наборе символов Unicode)	char x = 'c'; char y = '\x123'; char newline = '\n';
string	От пустой строки ("") до очень большого количества символов из набора Unicode	string s = "my name"; string empty = "";
bool	true и false	bool b = true;

Управление потоком выполнения

```
if (i < 10)
{
   // Выполняется, если i < 10
}
else
{
   // Выполняется в противном случае
}

while(i < 10)
{
   // Цикл выполняется, пока i < 10
}

for(int i = 0; i < 10; i++)
{
   // Выполнение 10 итераций
}

foreach(MyClass mc in myCollection)
{
   // Выполняется по одному разу для каждого объекта mc из
   // коллекции myCollection
}
```