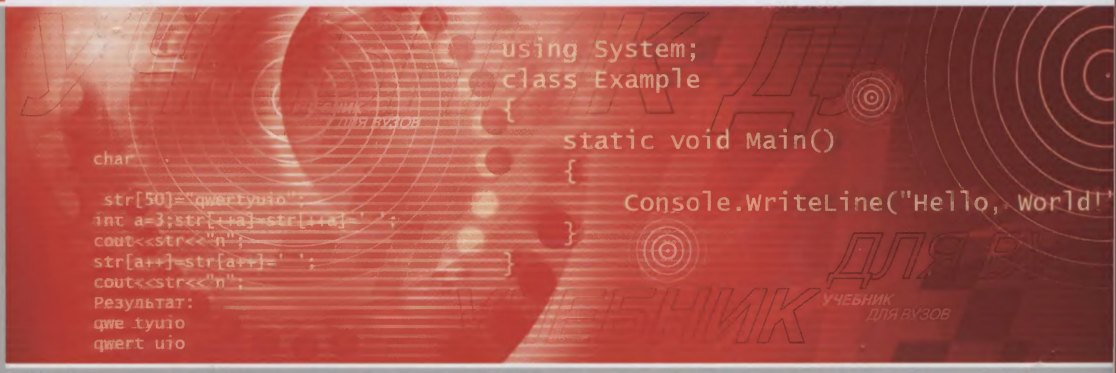
Учебник для вузов

ПИТЕР

Т. А. Павловская



C#

Программирование на языке высокого уровня

ДОПУЩЕНО
МИНИСТЕРСТВОМ ОБРАЗОВАНИЯ И НАУКИ РФ

Т. А. Павловская

C#

Программирование на языке высокого уровня

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

Павловская Татьяна Александровна

C#. Программирование на языке высокого уровня

Учебник для вузов

Рецензенты: Смирнова Н. Н. — заведующая кафедрой вычислительной техники Балтийского государственного технического университета «Военмех» им. Д. Ф. Устинова, кандидат технических наук, профессор;
Трофимов В. В. — заведующий кафедрой информатики Санкт-Петербургского государственного университета экономики и финансов, доктор технических наук, профессор

ББК 32.973-018.1
УДК 004.43

Павловская Т. А.

П12 C#. Программирование на языке высокого уровня: Учебник для вузов. — СПб.: Питер, 2014. — 432 с.: ил. — (Серия «Учебник для вузов»).

ISBN 978-5-496-00861-7

Задача этой книги — кратко, доступно и строго изложить основы C#, одного из самых перспективных современных языков программирования. Книга содержит описание версии C# 2.0 (2005) и предназначена для студентов, изучающих язык «с нуля», но будет полезна и опытным программистам, желающим освоить новый язык, не тратя времени на увесистые переводные фолианты.

Кроме конструкций языка в книге рассматриваются основные структуры данных, используемые при написании программ, классы библиотеки, а также рекомендации по стилю и технологии программирования. По ключевым темам приводятся задания для выполнения лабораторных работ, каждая из которых содержит по двадцать однотипных вариантов в расчете на учебную группу студентов.

Язык C# как средство обучения программированию обладает рядом несомненных достоинств. Он хорошо организован, строг, большинство его конструкций логичны и удобны, а развитые средства диагностики и редактирования кода делают процесс программирования приятным и эффективным.

Допущено Министерством образования и науки Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника».

ISBN 978-5-496-00861-7

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 17.09.13. Формат 70x100/16. Усл. п. л. 34,830. Тираж 1500. Заказ 7447

Отпечатано в ОАО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ». 432980, г. Ульяновск, ул. Гончарова, 14

Содержание

Предисловие

Задача этой книги — кратко, доступно и строго изложить основы C#, одного из самых перспективных современных языков программирования. Книга содержит описание версии C# 2.0 (2005) и предназначена для студентов, изучающих язык «с нуля», но будет полезна и опытным программистам, желающим освоить новый язык, не тратя времени на увесистые переводные фолианты.

Помимо конструкций языка в книге рассматриваются основные структуры данных, используемые при написании программ, классы библиотеки, а также рекомендации по стилю и технологии программирования. По ключевым темам приводятся задания для лабораторных работ, каждая из которых содержит до двадцати однотипных вариантов в расчете на учебную группу студентов.

Язык C# как средство обучения программированию обладает рядом несомненных достоинств. Он хорошо организован, строг, большинство его конструкций логичны и удобны. Развитые средства диагностики и редактирования кода делают процесс программирования приятным и эффективным. Мощная библиотека классов платформы .NET берет на себя массу рутинных операций, что дает возможность решать более сложные задачи, используя готовые «строительные блоки». Все это позволяет расценивать C# как перспективную замену языков Паскаль, BASIC и C++ при обучении программированию.

Немаловажно, что C# является не учебным, а профессиональным языком, предназначенным для решения широкого спектра задач, и в первую очередь — в быстро развивающейся области создания распределенных приложений. Поэтому базовый курс программирования, построенный на основе языка C#, позволит студентам быстрее стать востребованными специалистами-профессионалами.

Мощь языка C# имеет и оборотную сторону: во-первых, он достаточно требователен к ресурсам компьютера[1], во-вторых, для осмысленного написания простейшей программы, вычисляющей, «сколько будет дважды два», требуется изучить достаточно много материала, но многочисленные достоинства языка и платформы .NET перевешивают все недостатки.

Дополнительным достоинством является то, что компания Microsoft распространяет версию C# Express 2005, по которой можно познакомиться с языком бесплатно (http://msdn.microsoft.com/vstudio/express/visualCsharp/), что дает нам долгожданную возможность обойтись без пиратских копий программного обеспечения и почувствовать себя законопослушными гражданами.

---

[1] Например, минимальные требования для C# Express 2005 — Windows XP/2000, процессор Pentium 600 МГц и 128 Мбайт оперативной памяти.

ПРИМЕЧАНИЕ

Название языка вообще-то следовало бы произносить как «Си-диез». Компания Microsoft неоднократно подчеркивала, что, несмотря на некоторое различие в начертании и коде символов «решетки» и «диеза», в названии языка имеется в виду именно музыкальный диез и обыгрывается тот факт, что символом С в музыке обозначается нота «до». Таким образом, создатели языка вложили в его название смысл «на полтона выше С». Однако в нашей стране укоренилась калька с английского — «Си-шарп». Поклонники этого произношения должны иметь в виду, что язык C++ они должны называть тогда исключительно «Си-плас-плас», потому что программист прежде всего обязан следовать логике!

Все возможности языка C# описаны в соответствии с его спецификацией (http://www.ecma-international.org/publications/standards/Ecma-334.htm) и сопровождаются простыми примерами, к которым даны исчерпывающие пояснения. Большинство примеров представляют собой консольные приложения, однако в главе 14 дается введение в программирование под Windows, а в главе 15 — введение в создание веб-форм и веб-служб. Возможности, появившиеся во второй версии языка, рассматриваются не в отдельной главе, а по логике изложения материала, при этом они всегда снабжены соответствующей ремаркой для тех, кто пользуется платформой .NET предыдущих версий.

В книге приведено краткое введение в интересные и полезные темы, которые обычно не освещаются в базовом курсе C#, например, многопоточные приложения, асинхронные делегаты, асинхронный ввод-вывод и регулярные выражения. Определения синтаксических элементов языка выделены в тексте книги полужирным шрифтом. Все ключевые слова, типы, классы и большинство методов, описанных в книге, можно найти по алфавитному указателю, что позволяет использовать ее и в качестве справочника. Планируется интернет-поддержка книги на сайте http://ips.ifmo.ru в виде конспекта лекций, тестовых вопросов и проверки выполнения лабораторных работ.

Традиционного раздела «Благодарности» в этом предисловии не будет: при написании этой книги мне никто не помогал, и вся ответственность за ошибки лежит исключительно на мне. У меня нет собаки или жены, чтобы поблагодарить их, как это принято у зарубежных авторов, а мой попугай постоянно мешал мне своими громкими неорганизованными выкриками.

Ваши замечания, пожелания, дополнения, а также замеченные ошибки и опечатки не ленитесь присылать по адресу mux@tp2055.spb.edu — и тогда благодаря вам кто-то сможет получить более правильный вариант этой книги.

От издательства

Ваши замечания, предложения и вопросы отправляйте также по адресу электронной почты comp@piter.com (издательство «Питер»). Мы будем рады узнать ваше мнение! Подробную информацию о наших книгах вы найдете на веб-сайте издательства www.piter.com.

Глава 1
Первый взгляд на платформу .NET

Программист пишет программу, компьютер ее выполняет. Программа создается на языке, понятном человеку, а компьютер умеет исполнять только программы, написанные на его языке — в машинных кодах. Совокупность средств, с помощью которых программы пишут, корректируют, преобразуют в машинные коды, отлаживают и запускают, называют *средой разработки*, или *оболочкой*.

Среда разработки обычно содержит:

- текстовый *редактор*, предназначенный для ввода и корректировки текста программы;
- *компилятор*, с помощью которого программа переводится с языка, на котором она написана, в машинные коды;
- средства *отладки и запуска программ*;
- общие *библиотеки*, содержащие многократно используемые элементы программ;
- *справочную систему* и другие элементы.

Под *платформой* понимается нечто большее, чем среда разработки для одного языка. Платформа .NET (произносится «дотнет») включает не только среду разработки для нескольких языков программирования, называемую Visual Studio.NET, но и множество других средств, например, механизмы поддержки баз данных, электронной почты и коммерции.

В эпоху стремительного развития Интернета — глобальной информационной сети, объединяющей компьютеры разных архитектур, важнейшими задачами при создании программ становятся:

- *переносимость* — возможность выполнения на различных типах компьютеров;
- *безопасность* — невозможность несанкционированных действий;
- *надежность* — способность выполнять необходимые функции в предопределенных условиях; средний интервал между отказами;

- *использование готовых компонентов* — для ускорения разработки;
- *межъязыковое взаимодействие* — возможность применять одновременно несколько языков программирования.

Платформа .NET позволяет успешно решать все эти задачи. Для обеспечения переносимости компиляторы, входящие в состав платформы, переводят программу не в машинные коды, а в промежуточный язык (Microsoft Intermediate Language, MSIL, или просто IL), который не содержит команд, зависящих от языка, операционной системы и типа компьютера. Программа на этом языке выполняется не самостоятельно, а под управлением системы, которая называется *общеязыковой средой выполнения* (Common Language Runtime, CLR).

Среда CLR может быть реализована для любой операционной системы. При выполнении программы CLR вызывает так называемый JIT-компилятор, переводящий код с языка IL в машинные команды конкретного процессора, которые немедленно выполняются. JIT означает «just in time», что можно перевести как «вовремя», то есть компилируются только те части программы, которые требуется выполнить в данный момент. Каждая часть программы компилируется один раз и сохраняется в кэше[1] для дальнейшего использования. Схема выполнения программы при использовании платформы .NET приведена на рис. 1.1.

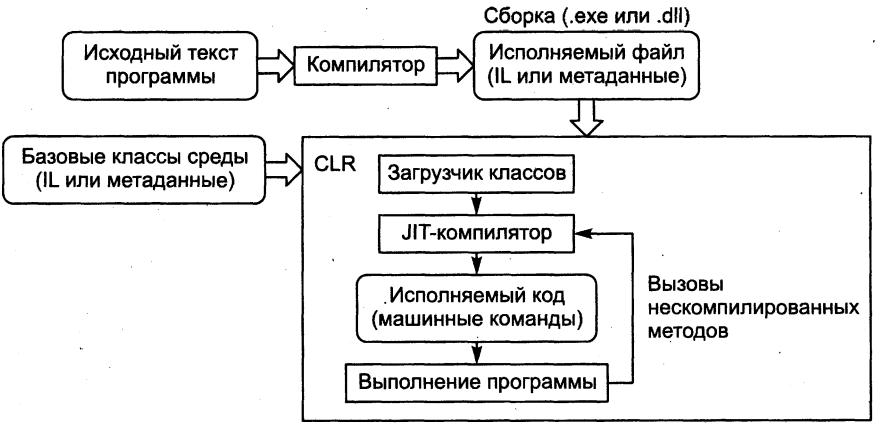


Рис. 1.1. Схема выполнения программы в .NET

Компилятор в качестве результата своего выполнения создает так называемую *сборку* — файл с расширением exe или dll, который содержит код на языке IL и метаданные. *Метаданные* представляют собой сведения об объектах, используемых в программе, а также сведения о самой сборке. Они позволяют организовать межъязыковое взаимодействие, обеспечивают безопасность и облегчают *развертывание приложений*, то есть установку программ на компьютеры пользователей.

[1] *Кэш* — область оперативной памяти, предназначенная для временного хранения информации.

ПРИМЕЧАНИЕ

Сборка может состоять из нескольких модулей. В любом случае она представляет собой программу, готовую для установки и не требующую для этого ни дополнительной информации, ни сложной последовательности действий. Каждая сборка имеет уникальное имя.

Во время работы программы среда CLR следит за тем, чтобы выполнялись только разрешенные операции, осуществляет распределение и очистку памяти и обрабатывает возникающие ошибки. Это многократно повышает безопасность и надежность программ.

Платформа .NET содержит огромную *библиотеку классов*[1], которые можно использовать при программировании на любом языке .NET. Общая структура библиотеки приведена на рис. 1.2. Библиотека имеет несколько уровней. На самом нижнем находятся *базовые классы среды*, которые используются при создании любой программы: классы ввода-вывода, обработки строк, управления безопасностью, графического интерфейса пользователя, хранения данных и пр.

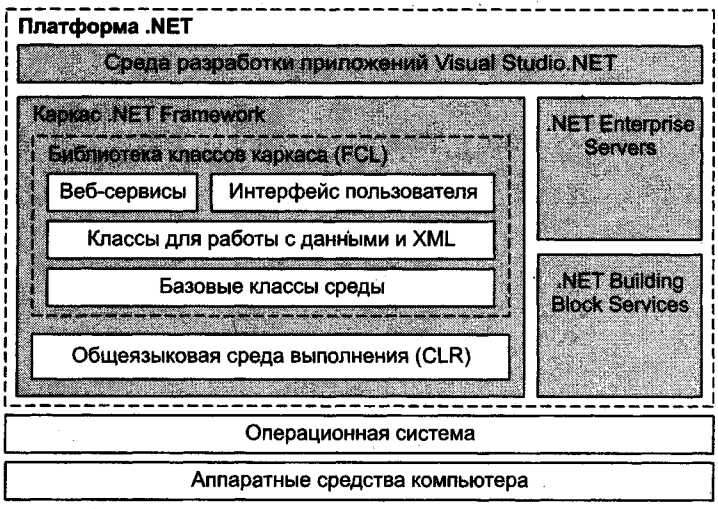


Рис. 1.2. Структура платформы .NET

Над этим слоем находится набор классов, позволяющий работать с базами данных и XML (с XML вы познакомитесь в последней главе этой книги). Классы самого верхнего уровня поддерживают разработку распределенных приложений, а также веб- и Windows-приложений. Программа может использовать классы любого уровня.

Подробное изучение библиотеки классов .NET — необходимая, но и наиболее трудоемкая задача программиста при освоении этой платформы. Библиотека

[1] Понятие класса рассматривается в следующем разделе. Пока можно считать, что класс служит для реализации некоторых функций.

классов вместе с CLR образуют *каркас* (framework), то есть основу платформы. Назначение остальных частей платформы мы рассмотрим по мере изучения материала.

ПРИМЕЧАНИЕ

Термин «приложение» можно для начала воспринимать как синоним слова «программа». Например, вместо фразы «программа, работающая под управлением Windows», говорят «Windows-приложение» или просто «приложение».

Платформа .NET рассчитана на объектно-ориентированную технологию создания программ, поэтому прежде чем начинать изучение языка C#, необходимо познакомиться с основными понятиями объектно-ориентированного программирования (ООП).

Объектно-ориентированное программирование

Принципы ООП проще всего понять на примере программ моделирования. В реальном мире каждый предмет или процесс обладает набором статических и динамических характеристик, иными словами, свойствами и поведением. *Поведение объекта* зависит от его *состояния* и *внешних воздействий*. Например, объект «автомобиль» никуда не поедет, если в баке нет бензина, а если повернуть руль, изменится положение колес.

Понятие объекта в программе совпадает с обыденным смыслом этого слова: *объект представляется как совокупность данных, характеризующих его состояние, и функций их обработки, моделирующих его поведение*. Вызов функции на выполнение часто называют *посылкой сообщения* объекту[1].

При создании объектно-ориентированной программы предметная область представляется в виде совокупности объектов. Выполнение программы состоит в том, что объекты обмениваются сообщениями. Это позволяет использовать при программировании понятия, более адекватно отражающие предметную область.

При представлении реального объекта с помощью программного необходимо выделить в первом его существенные особенности. Их список зависит от цели моделирования. Например, объект «крыса» с точки зрения биолога, изучающего миграции, ветеринара или, скажем, повара будет иметь совершенно разные характеристики. Выделение существенных с той или иной точки зрения свойств называется *абстрагированием*. Таким образом, программный объект — это абстракция.

Важным свойством объекта является его обособленность. Детали реализации объекта, то есть внутренние структуры данных и алгоритмы их обработки, скрыты

[1] Например, вызов функции «повернуть руль» интерпретируется как посылка сообщения «автомобиль, поверни руль!».

от пользователя объекта и недоступны для непреднамеренных изменений. Объект используется через его *интерфейс* — совокупность правил доступа.

Скрытие деталей реализации называется *инкапсуляцией* (от слова «капсула»). Ничего сложного в этом понятии нет: ведь и в обычной жизни мы пользуемся объектами через их интерфейсы. Сколько информации пришлось бы держать в голове, если бы для просмотра новостей надо было знать устройство телевизора!

Таким образом, объект является «черным ящиком», замкнутым по отношению к внешнему миру. Это позволяет представить программу в укрупненном виде — на уровне объектов и их взаимосвязей, а следовательно, управлять большим объемом информации и успешно отлаживать сложные программы.

Сказанное можно сформулировать более кратко и строго: *объект — это инкапсулированная абстракция с четко определенным интерфейсом.*

Инкапсуляция позволяет изменить реализацию объекта без модификации основной части программы, если его интерфейс остался прежним. Простота модификации является очень важным критерием качества программы, ведь любой программный продукт в течение своего жизненного цикла претерпевает множество изменений и дополнений.

Кроме того, инкапсуляция позволяет использовать объект в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти, а также создавать библиотеки объектов для применения во многих программах.

Каждый год в мире пишется огромное количество новых программ, и важнейшее значение приобретает возможность многократного использования кода. Преимущество объектно-ориентированного программирования состоит в том, что для объекта можно определить наследников, корректирующих или дополняющих его поведение. При этом нет необходимости не только повторять исходный код родительского объекта, но даже иметь к нему доступ.

Наследование является мощнейшим инструментом ООП и применяется для следующих взаимосвязанных целей:

- исключения из программы повторяющихся фрагментов кода;
- упрощения модификации программы;
- упрощения создания новых программ на основе существующих.

Кроме того, только благодаря наследованию появляется возможность использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

Наследование позволяет создавать *иерархии объектов*. Иерархия представляется в виде дерева, в котором более общие объекты располагаются ближе к корню, а более специализированные — на ветвях и листьях. Наследование облегчает использование библиотек объектов, поскольку программист может взять за основу объекты, разработанные кем-то другим, и создать наследников с требуемыми свойствами.

Объект, на основании которого строится новый объект, называется *родительским* объектом, объектом-предком, базовым классом, или суперклассом, а унаследованный от него объект — *потомком*, подклассом, или производным классом.

ООП позволяет писать гибкие, расширяемые и читабельные программы. Во многом это обеспечивается благодаря *полиморфизму*, под которым понимается возможность во время выполнения программы с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа. Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов, который мы рассмотрим в главе 8.

Подводя итог сказанному, сформулирую *достоинства ООП*:

- использование при программировании понятий, близких к предметной области;
- возможность успешно управлять большими объемами исходного кода благодаря инкапсуляции, то есть скрытию деталей реализации объектов и упрощению структуры программы;
- возможность многократного использования кода за счет наследования;
- сравнительно простая возможность модификации программ;
- возможность создания и использования библиотек объектов.

Эти преимущества особенно явно проявляются при разработке программ большого объема и классов программ. Однако ничто не дается даром: создание объектно-ориентированной программы представляет собой весьма непростую задачу, поскольку требует разработки иерархии объектов, а плохо спроектированная иерархия может свести к нулю все преимущества объектно-ориентированного подхода.

Кроме того, идеи ООП не просты для понимания и в особенности для практического применения. Чтобы эффективно использовать готовые объекты из библиотек, необходимо освоить большой объем достаточно сложной информации. Неграмотное же применение ООП способно привести к созданию излишне сложных программ, которые невозможно отлаживать и усовершенствовать.

Классы

Для представления объектов в языках C#, Java, C++, Delphi и т. п. используется понятие *класс*, аналогичное обыденному смыслу этого слова в контексте «класс членистоногих», «класс млекопитающих», «класс задач» и т. п. Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых *экземплярами класса*.

«Классический» класс содержит *данные*, задающие свойства объектов класса, и *функции*, определяющие их поведение. В последнее время в класс часто добавляется третья составляющая — *события*, на которые может реагировать объект класса[1].

Все классы библиотеки .NET, а также все классы, которые создает программист в среде .NET, имеют одного общего предка — класс `object` и организованы в единую иерархическую структуру. Внутри нее классы логически сгруппированы в так называемые *пространства имен*, которые служат для упорядочивания имен

[1] Это оправдано для классов, использующихся в программах, построенных на основе событийно-управляемой модели, например, при программировании для Windows.

классов и предотвращения конфликтов имен: в разных пространствах имена могут совпадать. Пространства имен могут быть вложенными, их идея аналогична знакомой вам иерархической структуре каталогов на компьютере.

Любая программа, создаваемая в .NET, использует пространство имен `System`. В нем определены классы, которые обеспечивают базовую функциональность, например, поддерживают выполнение математических операций, управление памятью и ввод-вывод.

Обычно в одно пространство имен объединяют взаимосвязанные классы. Например, пространство `System.Net` содержит классы, относящиеся к передаче данных по сети, `System.Windows.Forms` — элементы графического интерфейса пользователя, такие как формы, кнопки и т. д. Имя каждого пространства имен представляет собой неделимую сущность, однозначно его определяющую.

Последнее, о чем необходимо поговорить, прежде чем начать последовательное изучение языка C#, — среда разработки Visual Studio.NET.

Среда Visual Studio.NET

Среда разработки Visual Studio.NET предоставляет мощные и удобные средства написания, корректировки, компиляции, отладки и запуска приложений, использующих .NET-совместимые языки. Корпорация Microsoft включила в платформу средства разработки для четырех языков: C#, VB.NET, C++ и J#.

Платформа .NET является открытой средой. Это значит, что компиляторы для нее могут поставляться и сторонними разработчиками. К настоящему времени разработаны десятки компиляторов для .NET, например, Ada, COBOL, Delphi, Eiffel, Fortran, Lisp, Oberon, Perl и Python.

Все .NET-совместимые языки должны отвечать требованиям *общеязыковой спецификации* (Common Language Specification, CLS), в которой описывается набор общих для всех языков характеристик. Это позволяет использовать для разработки приложения несколько языков программирования и вести полноценную межъязыковую отладку. Все программы независимо от языка используют одни и те же базовые классы библиотеки .NET.

Приложение в процессе разработки называется *проектом*. Проект объединяет все необходимое для создания приложения: файлы, папки, ссылки и прочие ресурсы. Среда Visual Studio.NET позволяет создавать проекты различных типов, например:

- *Windows-приложение* использует элементы интерфейса Windows, включая формы, кнопки, флажки и пр.;
- *консольное приложение* выполняет вывод «на консоль», то есть в окно командного процессора;
- *библиотека классов* объединяет классы, которые предназначены для использования в других приложениях;

- *веб-приложение* — это приложение, доступ к которому выполняется через браузер (например, Internet Explorer) и которое по запросу формирует веб-страницу и отправляет ее клиенту по сети;
- *веб-сервис* — компонент, методы которого могут вызываться через Интернет.

Несколько проектов можно объединить в *решение* (solution). Это облегчает совместную разработку проектов.

Консольные приложения

Среда Visual Studio.NET работает на платформе Windows и ориентирована на создание Windows- и веб-приложений, однако разработчики предусмотрели работу и с консольными приложениями. При запуске консольного приложения операционная система создает так называемое *консольное окно*, через которое идет весь ввод-вывод программы. Внешне это напоминает работу в операционной системе в режиме *командной строки*, когда ввод-вывод представляет собой поток символов.

Консольные приложения наилучшим образом подходят для изучения языка, так как в них не используется множество стандартных объектов, необходимых для создания графического интерфейса. В первой части курса мы будем создавать только консольные приложения, чтобы сосредоточить внимание на базовых свойствах языка C#. В следующем разделе рассмотрены самые простые действия в среде: создание и запуск на выполнение консольного приложения на C#. Более полные сведения, необходимые для работы в Visual Studio.NET, можно получить из документации или книг [8], [16].

ПРИМЕЧАНИЕ

Большинство примеров, приведенных в книге, иллюстрируют базовые возможности C# и разрабатывались в интегрированной среде версии 7.1 (библиотека .NET Framework 1.1), однако вы можете работать и в более новых версиях. Программы, которые используют новые средства языка, появившиеся в спецификации версии 2.0, проверялись в Visual C# 2005 Express Edition (библиотека .NET Framework 2.0).

Создание проекта. Основные окна среды

Для создания проекта следует после запуска Visual Studio.NET[1] в главном меню выбрать команду File ▸ New ▸ Project.... В левой части открывшегося диалогового окна нужно выбрать пункт Visual C# Projects, в правой — пункт Console Application. В поле Name можно ввести имя проекта, а в поле Location — место его сохранения на диске, если заданные по умолчанию значения вас не устраивают. После щелчка на кнопке OK среда создаст решение и проект с указанным именем. Примерный вид экрана приведен на рис. 1.3.

В верхней части экрана располагается *главное меню* (с разделами File, Edit, View и т. д.) и *панели инструментов* (toolbars). Панелей инструментов в среде великое множество, и если включить их все (View ▸ Toolbars...), они займут половину экрана.

[1] Полезно создать для этого ярлык на рабочем столе или на панели быстрого запуска.

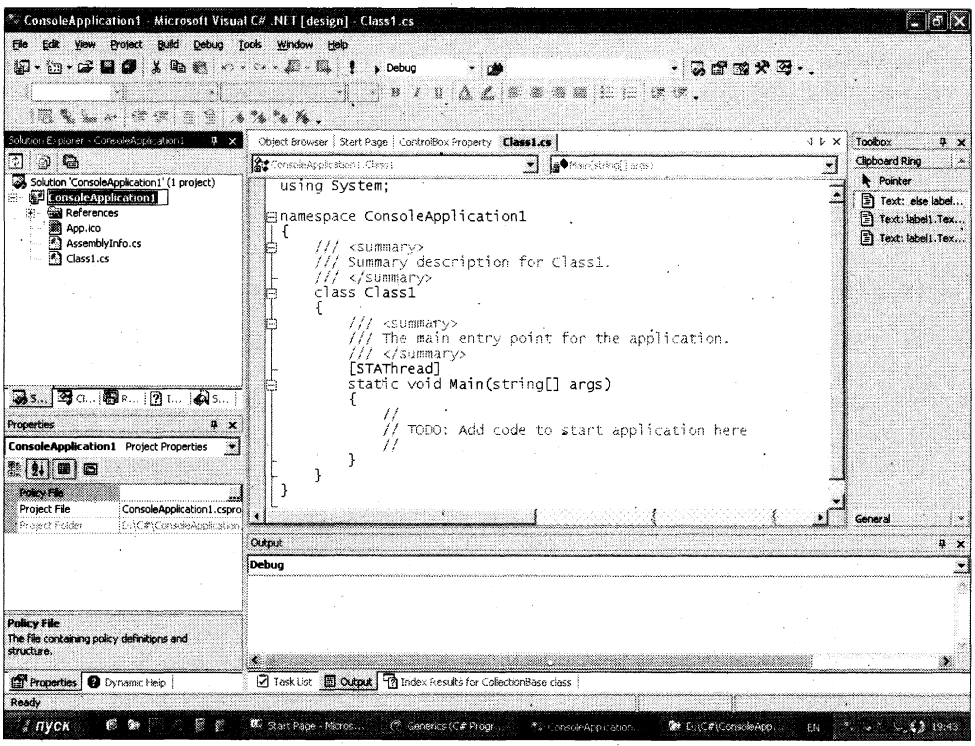


Рис. 1.3. Примерный вид экрана после создания проекта консольного приложения

В верхней левой части экрана располагается *окно управления проектом* Solution Explorer (если оно не отображается, следует воспользоваться командой View ▸ Solution Explorer главного меню). В окне перечислены все ресурсы, входящие в проект: ссылки на библиотеку (System, System.Data, System.XML), файл ярлыка (App.ico), файл с исходным текстом класса (Class1.cs) и информация о сборке[1] (AssemblyInfo.cs).

В этом же окне можно увидеть и другую информацию, если перейти на вкладку Class View, ярлычок которой находится в нижней части окна. На вкладке Class View представлен список всех классов, входящих в приложение, их элементов и предков.

ПРИМЕЧАНИЕ

Небезынтересно полюбопытствовать, какие файлы создала среда для поддержки проекта. С помощью проводника Windows можно увидеть, что на заданном диске появилась папка с указанным именем, содержащая несколько других файлов и вложенных папок. Среди них — файл проекта (с расширением csproj), файл решения (с расширением sln) и файл с кодом класса (Class1.cs).

В нижней левой части экрана расположено *окно свойств* Properties (если окна не видно, воспользуйтесь командой View ▸ Properties главного меню). В окне свойств

[1] Как говорилось в предыдущем разделе, сборка является результатом работы компилятора и содержит код на промежуточном языке и метаданные.

отображаются важнейшие характеристики выделенного элемента. Например, чтобы изменить имя файла, в котором хранится класс Class1, надо выделить этот файл в окне управления проектом и задать в окне свойств новое значение свойства FileName (ввод заканчивается нажатием клавиши Enter).

Основное пространство экрана занимает *окно редактора*, в котором располагается текст программы, созданный средой автоматически. Текст представляет собой каркас, в который программист добавляет код по мере необходимости.

Ключевые (зарезервированные) слова[1] отображаются синим цветом, комментарии[2] различных типов — серым и темно-зеленым, остальной текст — черным. Слева от текста находятся *символы структуры*: щелкнув на любом квадратике с минусом, можно скрыть соответствующий блок кода. При этом минус превращается в плюс, щелкнув на котором, можно опять вывести блок на экран. Это средство хорошо визуально структурирует код и позволяет сфокусировать внимание на нужных фрагментах.

Заготовка консольной программы

Рассмотрим каждую строку заготовки программы (листинг 1.1). Не пытайтесь сразу понять абсолютно все, что в ней написано. Пока что ваша цель — изучить принципы работы в оболочке, а досконально разбираться в программе мы будем позже.

Листинг 1.1. Заготовка консольной программы

```
using System;

namespace ConsoleApplication1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application her
            //
        }
    }
}
```

[1] Ключевые слова — это слова, имеющие специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены.

[2] Комментарии предназначены для программиста и позволяют обеспечить документирование программы.

Директива `using System` разрешает использовать имена стандартных классов из пространства имен `System` непосредственно (без указания имени пространства).

Ключевое слово `namespace` создает для проекта собственное пространство имен, названное по умолчанию `ConsoleApplication1`. Это сделано для того, чтобы можно было давать программным объектам имена, не заботясь о том, что они могут совпасть с именами в других пространствах имен.

Строки, начинающиеся с двух или трех косых черт, являются комментариями и предназначены для документирования текста программы.

C# — объектно-ориентированный язык, поэтому написанная на нем программа представляет собой совокупность взаимодействующих между собой классов. В нашей заготовке программы всего один класс, которому по умолчанию задано имя `Class1`. *Описание класса* начинается с ключевого слова `class`, за которым следуют его имя и далее в фигурных скобках — список элементов класса (его данных и функций, называемых также методами).

ВНИМАНИЕ

Фигурные скобки являются важным элементом синтаксиса. Каждой открывающей скобке соответствует своя закрывающая, которая обычно располагается ниже по тексту с тем же отступом. Эти скобки ограничивают *блок*, внутри которого могут располагаться другие блоки, вложенные в него, как матрешки. Блок может применяться в любом месте, где допускается отдельный оператор.

В данном случае внутри класса только один элемент — метод `Main`. Каждое приложение должно содержать метод `Main` — с него начинается выполнение программы. Все методы описываются по единым правилам.

Упрощенный синтаксис метода:

```
[ спецификаторы ] тип имя_метода ( [ параметры ] )
{
    тело метода: действия, выполняемые методом
}
```

ВНИМАНИЕ

Для описания языка программирования в документации часто используется некоторый формальный метаязык, например, формулы Бэкуса—Наура или синтаксические диаграммы. Для наглядности и простоты изложения в этой книге используется широко распространенный неформальный способ описания, когда необязательные части синтаксических конструкций заключаются в квадратные скобки, текст, который необходимо заменить конкретным значением, пишется по-русски, а возможность выбора одного из нескольких элементов обозначается вертикальной чертой. Например:

```
[ void | int ] имя_метода();
```

Эта запись означает, что вместо конструкции `имя_метода` необходимо указать конкретное имя в соответствии с правилами языка, а перед ним может находиться либо слово `void`, либо слово `int`, либо ничего. Символ подчеркивания используется для связи слов вместо пробела, показывая, что на этом месте должен стоять один синтаксический элемент, а не два. В тех случаях, когда квадратные скобки являются элементом синтаксиса, это оговаривается особо.

Таким образом, любой метод должен иметь *тип*, *имя* и *тело*, остальные части описания являются необязательными, поэтому мы их пока проигнорируем. Методы подробно рассматриваются в разделе «Методы» (см. с. 106).

ПРИМЕЧАНИЕ

Наряду с понятием «метод» часто используется другое — функция-член класса. Метод является частным случаем *функции* — законченного фрагмента кода, который можно вызвать по имени. Далее в книге используются оба эти понятия.

Среда заботливо поместила внутрь метода `Main` комментарий:

```
// TODO: Add code to start application here
```

Это означает: «Добавьте сюда код, выполняемый при запуске приложения». Последуем совету и добавим после строк комментария (но не в той же строке!) строку

```
Console.WriteLine("Ур-ра! Зар-работало! (c) Кот Матроскин");
```

Здесь `Console` — это имя стандартного класса из пространства имен `System`. Его метод `WriteLine` выводит на экран заданный в кавычках текст. Как видите, для обращения к методу класса используется конструкция

```
имя_класса.имя_метода
```

Если вы не сделали ошибку в первом же слове, то сразу после ввода с клавиатуры следом за словом `Console` символа точки среда выведет подсказку, содержащую список всех доступных элементов класса `Console`. Выбор нужного имени выполняется либо мышью, либо клавишами управления курсором, либо вводом одного или нескольких начальных символов имени. При нажатии клавиши Enter выбранное имя появляется в тексте программы.

СОВЕТ

Не пренебрегайте возможностью автоматического ввода — это убережет вас от опечаток и сэкономит время. Если подсказка не появляется, это свидетельствует об ошибке в имени или в месте расположения в программе вводимого текста.

Программа должна приобрести вид, приведенный в листинге 1.2 (для того чтобы вы могли сосредоточиться на структуре программы, из нее убраны все комментарии и другие пока лишние для нас детали). Ничего ужасного в этом листинге нет, не правда ли?

Обратите внимание на то, что после внесения изменений около имени файла на ярлычке в верхней части окна редактора появился символ * — это означает, что текст, сохраненный на диске, и текст, представленный в окне редактора, не совпадают. Для сохранения файла воспользуйтесь командой File ▸ Save главного меню или кнопкой Save на панели инструментов (текстовый курсор должен при этом находиться в окне редактора). Впрочем, при запуске программы среда сохранит исходный текст самостоятельно.

Листинг 1.2. Первая программа на C#

```
using System;
namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            Console.WriteLine( "Ур-ра! Зар-работало! (c) Кот Матроскин" );
        }
    }
}
```

На рис. 1.4 приведен экран после создания консольного приложения в Visual C# 2005 Express Edition. Как видите, текст заготовки приложения более лаконичен, чем в предыдущей версии, и практически совпадает с листингом 1.1 без учета комментариев.

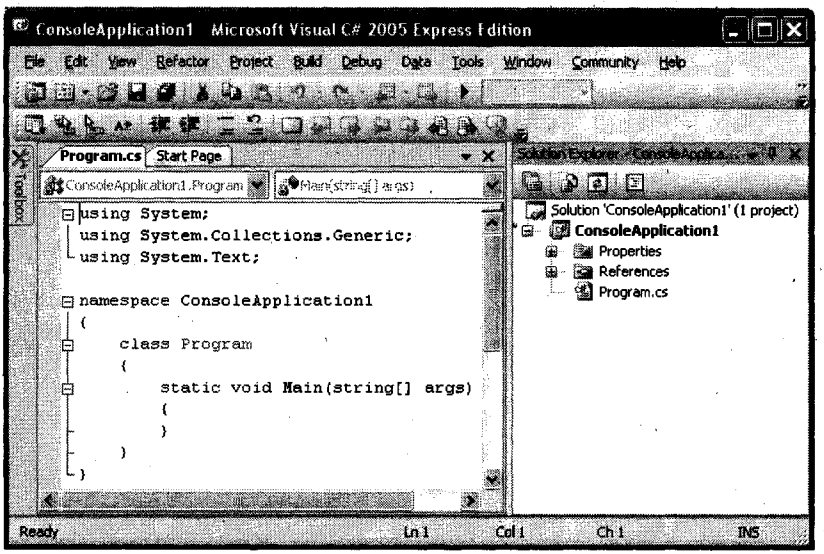


Рис. 1.4. Вид экрана после создания проекта консольного приложения в Visual C# 2005 Express Edition

Запуск программы

Самый простой способ запустить программу — нажать клавишу F5 (или выбрать в меню команду Debug ▸ Start). Если программа написана без ошибок, то фраза Ур-ра! Зар-работало! (c) Кот Матроскин промелькнет перед вашими глазами в консольном окне, которое незамедлительно закроется. Это хороший результат, но для того чтобы понаблюдать его спокойно, следует воспользоваться клавишами Ctrl+F5[1] (или выбрать в меню команду Debug ▸ Start Without Debugging).

[1] Обозначение Ctrl+F5 означает, что следует, удерживая клавишу Ctrl, нажать F5.

После внесения изменений компилятор может обнаружить в тексте программы *синтаксические ошибки*. Он сообщает об этом в окне, расположенном в нижней части экрана.

Давайте внесем в программу синтаксическую ошибку, чтобы впервые увидеть то, что впоследствии вам придется наблюдать многие тысячи раз. Уберите точку с запятой после оператора `Console.WriteLine(...)` и запустите программу заново. Вы увидите диалоговое окно с сообщением о том, что при построении приложения обнаружены ошибки, и вопросом, продолжать ли дальше (There were build errors. Continue?). Сурово ответив No, обратимся к окну вывода ошибок за разъяснениями.

ПРИМЕЧАНИЕ

Если сообщения об ошибке не видно, просмотрите содержимое окна с помощью полосы прокрутки, пытаясь найти в нем слово «error».

Двойной щелчок на строке с сообщением об ошибке error CS1002 ; expected (означающее, что здесь ожидалась точка с запятой) подсвечивает неверную строку в программе. Для получения дополнительных пояснений можно нажать клавишу F1, при этом в окне редактора появится новая вкладка с соответствующей страницей из справочной системы. Для возвращения к коду класса следует щелкнуть на ярлычке с именем файла в верхней строке редактора.

Другой способ получения справки — использовать окно Dynamic Help, расположенное на том же месте, что и окно свойств Properties (его ярлычок находится в нижней строке окна). Содержимое этого окна динамически меняется в зависимости от того, какой элемент выделен. Для получения справки надо щелкнуть на соответствующей ссылке, и страница справочной системы будет отображена на отдельной вкладке окна редактора (в Visual C# 2005 Express Edition справочная информация появляется в отдельном окне).

Теперь, когда вы получили общее представление о платформе .NET, можно, наконец, приступить к планомерному изучению языка C#.

Рекомендации по программированию

В этой главе дается очень краткое введение в интересную и обширную тему — платформу .NET. Для более глубокого понимания механизмов ее функционирования настоятельно рекомендуется изучить дополнительную литературу (например, [5], [19], [26], [27]) и публикации в Интернете.

Среда разработки Visual Studio.NET предоставляет программисту мощные и удобные средства написания, корректировки, компиляции, отладки и запуска приложений, описанию которых посвящены целые книги. В процессе изучения языка C# желательно постепенно изучать эти возможности, ведь чем лучше вы будете владеть инструментом, тем эффективнее и приятнее будет процесс программирования.

Глава 2
Основные понятия языка

В этой главе рассматриваются элементарные «строительные блоки» языка C#, вводится понятие типа данных и приводится несколько классификаций типов.

Состав языка

Язык программирования можно уподобить очень примитивному иностранному языку с жесткими правилами без исключений. Изучение иностранного языка обычно начинают с алфавита, затем переходят к словам и законам построения фраз, и только в результате длительной практики и накопления словарного запаса появляется возможность свободно выражать на этом языке свои мысли. Примерно так же поступим и мы при изучении языка C#.

Алфавит и лексемы

Все тексты на языке пишутся с помощью его *алфавита*. Например, в русском языке один алфавит (набор символов), а в албанском — другой. В C# используется кодировка символов Unicode. Давайте разберемся, что это такое.

Компьютер умеет работать только с числами, и для того чтобы можно было хранить в его памяти текст, требуется определить, каким числом будет представляться (кодироваться) каждый символ. Соответствие между символами и кодирующими их числами называется *кодировкой*, или *кодовой таблицей* (character set).

Существует множество различных кодировок символов. Например, в Windows часто используется кодировка ANSI, а конкретно — CP1251. Каждый символ представляется в ней одним байтом (8 бит), поэтому в этой кодировке можно одновременно задать только 256 символов. В первой половине кодовой таблицы находятся латинские буквы, цифры, знаки арифметических операций и другие распространенные символы. Вторую половину занимают символы русского алфа-

вита. Если требуется представлять символы другого национального алфавита (например, албанского), необходимо использовать другую кодовую таблицу.

Кодировка Unicode позволяет представить символы всех существующих алфавитов одновременно, что коренным образом улучшает переносимость текстов. Каждому символу соответствует свой уникальный код. Естественно, что при этом для хранения каждого символа требуется больше памяти. Первые 128 Unicode-символов соответствуют первой части кодовой таблицы ANSI.

Алфавит C# включает:

- *буквы* (латинские и национальных алфавитов) и *символ подчеркивания* (_), который употребляется наряду с буквами;
- *цифры*;
- *специальные символы*, например +, *, { и &;
- *пробельные символы* (пробел и символы табуляции);
- *символы перевода строки*.

Из символов составляются более крупные строительные блоки: лексемы, директивы препроцессора и комментарии.

Лексема (token[1]) —это минимальная единица языка, имеющая самостоятельный смысл. Существуют следующие виды лексем:

- *имена* (*идентификаторы*);
- *ключевые слова*;
- *знаки операций*;
- *разделители*;
- *литералы* (*константы*).

Лексемы языка программирования аналогичны словам естественного языка. Например, лексемами являются число 128 (но не его часть 12), имя `Vasia`, ключевое слово `goto` и знак операции сложения +. Далее мы рассмотрим лексемы подробнее.

Директивы препроцессора пришли в C# из его предшественника — языка C++. Препроцессором называется предварительная стадия компиляции, на которой формируется окончательный вид исходного текста программы. Например, с помощью директив (инструкций, команд) препроцессора можно включить или выключить из процесса компиляции фрагменты кода. Директивы препроцессора не играют в C# такой важной роли, как в C++. Мы рассмотрим их в свое время — в разделе «Директивы препроцессора» (см. с. 287).

Комментарии предназначены для записи пояснений к программе и формирования документации. Правила записи комментариев описаны далее в этом разделе.

Из лексем составляются выражения и операторы. *Выражение* задает правило вычисления некоторого значения. Например, выражение `a + b` задает правило вычисления суммы двух величин.

[1] Часто это слово ленятся переводить и пишут просто «токен».

ПРИМЕЧАНИЕ

Компилятор выполняет перевод программы в исполняемый файл на языке IL за две последовательные фазы: лексического и синтаксического анализа. При лексическом анализе из потока символов, составляющих исходный текст программы, выделяются лексемы (по другим лексемам, разделителям, пробельным символам и переводам строки). На этапе синтаксического анализа программа переводится в исполняемый код. Кроме того, компилятор формирует сообщения о синтаксических ошибках.

Оператор задает законченное описание некоторого действия, данных или элемента программы. Например:

```
int a;
```

Это — оператор описания целочисленной переменной `a`.

Идентификаторы

Имена в программах служат той же цели, что и имена в мире людей, — чтобы обращаться к программным объектам и различать их, то есть идентифицировать. Поэтому имена также называют *идентификаторами*. В идентификаторе могут использоваться буквы, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, `sysop`, `SySoP` и `SYSOP` — три разных имени.

Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Длина идентификатора не ограничена. Пробелы внутри имен не допускаются.

В идентификаторах C# разрешается использовать помимо латинских букв буквы национальных алфавитов. Например, `Пёсик` или `ээ` являются правильными идентификаторами[1]. Более того, в идентификаторах можно применять даже так называемые *escape-последовательности Unicode*, то есть представлять символ с помощью его кода в шестнадцатеричном виде с префиксом `\u`, например, `\u00F2`.

ПРИМЕЧАНИЕ

Примеры неправильных имен: `2late`, `Big gig`, `Б#г`; первое начинается с цифры, второе и третье содержат недопустимые символы (пробел и #).

Имена даются элементам программы, к которым требуется обращаться: переменным, типам, константам, методам, меткам и т. д. Идентификатор создается на этапе объявления переменной (метода, типа и т. п.), после этого его можно использовать в последующих операторах программы. При выборе идентификатора необходимо иметь в виду следующее:

- идентификатор не должен совпадать с ключевыми словами (см. следующий раздел)[2];

[1] Другое дело, захочется ли вам при вводе текста программы все время переключать регистр. Пример программы, в которой в именах используется кириллица, приведен на с. 182.

[2] Впрочем, для использования ключевого слова в качестве идентификатора его достаточно предварить символом `@`. Например, правильным будет идентификатор `@if`.

- не рекомендуется начинать идентификаторы с двух символов подчеркивания, поскольку такие имена зарезервированы для служебного использования.

Для улучшения читабельности программы следует давать объектам осмысленные имена, составленные в соответствии с определенными правилами. Понятные и согласованные между собой имена — основа хорошего стиля программирования. Существует несколько видов так называемых *нотаций* — соглашений о правилах создания имен.

В *нотации Паскаля* каждое слово, составляющее идентификатор, начинается с прописной буквы, например, `MaxLength`, `MyFuzzyShooshpanchik`.

Венгерская нотация (ее предложил венгр по национальности, сотрудник компании Microsoft) отличается от предыдущей наличием префикса, соответствующего типу величины, например, `iMaxLength`, `lpfnMyFuzzyShooshpanchik`.

Согласно нотации *Camel*, с прописной буквы начинается каждое слово, составляющее идентификатор, кроме первого, например, `maxLength`, `myFuzzyShooshpanchik`. Человеку с богатой фантазией абрис имени может напоминать верблюда, откуда и произошло название этой нотации.

Еще одна традиция — разделять слова, составляющие имя, знаками подчеркивания: `max_length`, `my_fuzzy_shooshpanchik`, при этом все составные части начинаются со строчной буквы.

В C# для именования различных видов программных объектов чаще всего используются две нотации: Паскаля и Camel. Многобуквенные идентификаторы в примерах этой книги соответствуют рекомендациям, приведенным в спецификации языка. Кроме того, в примерах для краткости часто используются однобуквенные имена. В реальных программах такие имена можно применять только в ограниченном наборе случаев.

Ключевые слова

Ключевые слова — это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Список ключевых слов C# приведен в табл. 2.1.

Знаки операций и разделители

Знак операции — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Например, в выражении `a += b` знак `+=` является знаком операции, а `a` и `b` — операндами. Символы, составляющие знак операций, могут быть как специальными, например, `&&`, `|` и `<`, так и буквенными, такими как `as` или `new`.

Операции делятся на *унарные*, *бинарные* и *тернарную* по количеству участвующих в них операндов. Один и тот же знак может интерпретироваться по-разному в зависимости от контекста. Все знаки операций, за исключением `[ ]`, `( )` и `? :`, представляют собой отдельные лексемы.

Таблица 2.1. Ключевые слова C#

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

ПРИМЕЧАНИЕ

Знаки операций C# описаны в разделе «Операции и выражения» (см. с. 42). Большинство стандартных операций может быть переопределено (перегружено). Перегрузка операций рассматривается в разделе «Операции класса» (см. с. 161).

Разделители используются для разделения или, наоборот, группирования элементов. Примеры разделителей: скобки, точка, запятая. Ниже перечислены все знаки операций и разделители, использующиеся в C#:

```
{ } [ ] ( ) . , : ; + - * / % & | ^ ! ~ =
< > ? ++ -- && || << >> == != <= >= += -= *= /= %=
&= |= ^= <<= >>= ->
```

Литералы

Литералами, или *константами*, называют неизменяемые величины. В C# есть логические, целые, вещественные, символьные и строковые константы, а также константа `null`. Компилятор, выделив константу в качестве лексемы, относит ее к одному из типов данных по ее внешнему виду. Программист может задать тип константы и самостоятельно[1].

Описание и примеры констант каждого типа приведены в табл. 2.2. Примеры, иллюстрирующие наиболее часто употребляемые формы констант, выделены полужирным шрифтом (при первом чтении можно обратить внимание только на них).

[1] Определение типа будет введено чуть позже в этой главе, а пока можно использовать обыденное значение этого слова.

Таблица 2.2. Константы в C#

Константа	Описание	Примеры
Логическая	`true` (истина) или `false` (ложь)	`true` `false`
Целая	*Десятичная*: последовательность десятичных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), за которой может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu)	`8 0 199226` `8u 0Lu 199226L`

	Шестнадцатеричная: символы `0x` или `0X`, за которыми следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), а за цифрами, в свою очередь, может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu)	`0xA 0x1B8 0X00FF`
		`0xAU 0x1B8LU`
		`0X00FFl`

Вещественная	*С фиксированной точкой*[1]:
	[цифры][.][цифры][суффикс]
	Суффикс — один из символов F, f, D, d, M, m
		`5.7 .001 35`
		`5.7F .001d 35`
		`5F .001f 35m`

	С порядком:
	[цифры][.][цифры]{E|e}[+|–][цифры][суффикс]
	Суффикс — один из символов F, f, D, d, M, m
		`0.2E6 .11e+3 5E-10`
		`0.2E6D .11e-3`
		`5E10`

Символьная	Символ, заключенный в апострофы	`'A' 'ю' '*'`
		`'\0' '\n'`
		`'\xF' '\x74'`
		`'\uA81B'`

Строковая	Последовательность символов, заключенная в кавычки	`"Здесь был Vasia"`
		`"\t3начение r = \xF5 \n"`
		`"Здесь был \u0056\u0061"`
		`"C:\\temp\\file1.txt"`
		`@"C:\temp\file1.txt"`

Константа `null`	Ссылка, которая не указывает ни на какой объект	`null`

Рассмотрим табл. 2.2 более подробно. *Логических* литералов всего два. Они широко используются в качестве признаков наличия или отсутствия чего-либо.

Целые литералы могут быть представлены либо в десятичной, либо в шестнадцатеричной системе счисления, а *вещественные* — только в десятичной системе, но

[1] Напомню, что квадратные скобки при описании означают необязательность заключенной в них конструкции.

в двух формах: с фиксированной точкой и с порядком. Вещественная константа с порядком представляется в виде *мантиссы* и *порядка*. Мантисса записывается слева от знака экспоненты (E или e), порядок — справа от знака. Значение константы определяется как произведение мантиссы и возведенного в указанную в порядке степень числа 10 (например, 1.3e2 = 1,3 · 10^2 = 130). При записи вещественного числа целая часть может быть опущена, например, .1.

ВНИМАНИЕ

Пробелы внутри числа не допускаются. Для отделения целой части от дробной используется не запятая, а точка. Символ E не представляет собой знакомое всем из математики число *e*, а указывает, что далее располагается степень, в которую нужно возвести число 10.

Если требуется сформировать *отрицательную* целую или вещественную константу, то перед ней ставится знак унарной операции изменения знака (-), например: –218, –022, –0x3C, –4.8, –0.1e4.

Когда компилятор распознает константу, он отводит ей место в памяти в соответствии с ее видом и значением. Если по каким-либо причинам требуется явным образом задать, сколько памяти следует отвести под константу, используются *суффиксы*, описания которых приведены в табл. 2.3. Поскольку такая необходимость возникает нечасто, эту информацию можно при первом чтении пропустить.

Таблица 2.3. Суффиксы целых и вещественных констант

Суффикс	Значение
L, l	Длинное целое (long)
U, u	Беззнаковое целое (unsigned)
F, f	Вещественное с одинарной точностью (float)
D, d	Вещественное с двойной точностью (double)
M, m	Финансовое десятичного типа (decimal)

Допустимые диапазоны значений целых и вещественных констант в зависимости от префикса мы рассмотрим немного позже в этой главе.

Символьная константа представляет собой любой символ в кодировке Unicode. Символьные константы записываются в одной из четырех форм, представленных в табл. 2.2:

- «обычный» символ, имеющий графическое представление (кроме апострофа и символа перевода строки), — 'A', 'ю', '*';
- управляющая последовательность — '\0', '\n';
- символ в виде шестнадцатеричного кода — '\xF', '\x74';
- символ в виде escape-последовательности Unicode — '\uA81B'.

Управляющей последовательностью, или *простой escape-последовательностью*, называют определенный символ, предваряемый обратной косой чертой. Управляющая последовательность интерпретируется как одиночный символ и используется для представления:

- кодов, не имеющих графического изображения (например, `\n` — переход в начало следующей строки);
- символов, имеющих специальное значение в строковых и символьных литералах, например, апострофа `'`.

В табл. 2.4 приведены допустимые значения последовательностей. Если непосредственно за обратной косой чертой следует символ, не предусмотренный таблицей, возникает ошибка компиляции.

Таблица 2.4. Управляющие последовательности в C#

Вид	Наименование
`\a`	Звуковой сигнал
`\b`	Возврат на шаг
`\f`	Перевод страницы (формата)
`\n`	Перевод строки
`\r`	Возврат каретки
`\t`	Горизонтальная табуляция
`\v`	Вертикальная табуляция
`\\`	Обратная косая черта
`\'`	Апостроф
`\"`	Кавычка
`\0`	Нуль-символ

Символ, представленный в виде *шестнадцатеричного кода*, начинается с префикса `\0x`, за которым следует код символа. Числовое значение должно находиться в диапазоне от 0 до $2^{16} - 1$, иначе возникает ошибка компиляции.

Escape-последовательности Unicode служат для представления символа в кодировке Unicode с помощью его кода в шестнадцатеричном виде с префиксом `\u` или `\U`, например, `\u00F2`, `\U00010011`. Коды в диапазоне от `\U10000` до `\U10FFFF` представляются в виде двух последовательных символов; коды, превышающие `\U10FFFF`, не поддерживаются.

Управляющие последовательности обоих видов могут использоваться и в *строковых константах*, называемых иначе *строковыми литералами*. Например, если требуется вывести несколько строк, можно объединить их в один литерал, отделив одну строку от другой символами `\n`:

```
"Никто не доволен своей\nвнешностью, но каждый доволен\nсвоим умом"
```

Этот литерал при выводе будет выглядеть так:

```
Никто не доволен своей
внешностью, но каждый доволен
своим умом
```

Другой пример: если внутри строки требуется использовать кавычку, ее предваряют косой чертой, по которой компилятор отличает ее от кавычки, ограничивающей строку:

```
"Издательский дом \"Питер\""
```

Как видите, строковые литералы с управляющими символами несколько теряют в читабельности, поэтому в C# введен второй вид литералов — *дословные литералы* (verbatim strings). Эти литералы предваряются символом `@`, который отключает обработку управляющих последовательностей и позволяет получать строки в том виде, в котором они записаны. Например, два приведенных выше литерала в дословном виде выглядят так:

```
@"Никто не доволен своей
внешностью, но каждый доволен
своим умом"
@"Издательский дом "Питер""
```

Чаще всего дословные литералы применяются в регулярных выражениях и при задании полного пути файла, поскольку в нем присутствуют символы обратной косой черты, которые в обычном литерале пришлось бы представлять с помощью управляющей последовательности. Сравните два варианта записи одного и того же пути:

```
"C:\\app\\bin\\debug\\a.exe"
@"C:\app\bin\debug\a.exe"
```

Строка может быть пустой (записывается парой смежных двойных кавычек `""`), пустая символьная константа недопустима.

Константа `null` представляет собой значение, задаваемое по умолчанию для величин так называемых ссылочных типов, которые мы рассмотрим далее в этой главе.

Комментарии

Комментарии предназначены для записи пояснений к программе и формирования документации. Компилятор комментарии игнорирует. Внутри комментария можно использовать любые символы. В C# есть два вида комментариев: однострочные и многострочные.

Однострочный комментарий начинается с двух символов прямой косой черты (`//`) и заканчивается символом перехода на новую строку, *многострочный* заключается между символами-скобками `/*` и `*/` и может занимать часть строки, целую строку или несколько строк. Комментарии не вкладываются друг в друга: символы `//` и `/*` не обладают никаким специальным значением внутри комментария.

Кроме того, в языке есть еще одна разновидность комментариев, которые начинаются с трех подряд идущих символов косой черты (///). Они предназначены для формирования документации к программе в формате XML. Компилятор извлекает эти комментарии из программы, проверяет их соответствие правилам и записывает их в отдельный файл. Правила задания комментариев этого вида мы рассмотрим в главе 15.

Типы данных

Данные, с которыми работает программа, хранятся в оперативной памяти. Естественно, что компилятору необходимо точно знать, сколько места они занимают, как именно закодированы и какие действия с ними можно выполнять. Все это задается при описании данных с помощью типа.

Тип данных однозначно определяет:

- *внутреннее представление* данных, а следовательно, и *множество их возможных значений*;
- *допустимые действия* над данными (операции и функции).

Например, целые и вещественные числа, даже если они занимают одинаковый объем памяти, имеют совершенно разные диапазоны возможных значений; целые числа можно умножать друг на друга, а, например, символы — нельзя.

Каждое выражение в программе имеет определенный тип. Величин, не имеющих никакого типа, не существует. Компилятор использует информацию о типе при проверке допустимости описанных в программе действий.

Память, в которой хранятся данные во время выполнения программы, делится на две области: стек (stack) и динамическая область, или хип (heap)[1]. *Стек* используется для хранения величин, память под которые выделяет компилятор, а в *динамической области* память резервируется и освобождается во время выполнения программы с помощью специальных команд. Основным местом для хранения данных в C# является хип.

Классификация типов

Любая информация легче усваивается, если она «разложена по полочкам». Поэтому, прежде чем перейти к изучению конкретных типов языка C#, рассмотрим их классификацию. Типы можно классифицировать по разным признакам.

Если принять за основу строение элемента, все типы можно разделить на *простые* (не имеют внутренней структуры) и *структурированные* (состоят из элементов других типов). По своему «создателю» типы можно разделить на *встроенные*

[1] В русскоязычной литературе для этого термина часто используют романтичный синоним «куча». А в куче встречается «мусор», что означает не то, что вы могли подумать, а «неиспользуемые величины» — впрочем, пока мы оставим мусор без внимания!

(стандартные) и *определяемые программистом* (рис. 2.1). Для данных *статического* типа память выделяется в момент объявления, при этом ее требуемый объем известен. Для данных *динамического* типа размер данных в момент объявления может быть неизвестен, и память под них выделяется по запросу в процессе выполнения программы.

Рис. 2.1. Различные классификации типов данных C#

Встроенные типы

Встроенные типы не требуют предварительного определения. Для каждого типа существует ключевое слово, которое используется при описании переменных, констант и т. д. Если же программист определяет собственный тип данных, он описывает его характеристики и сам дает ему имя, которое затем применяется точно так же, как имена стандартных типов. Описание собственного типа данных должно включать всю информацию, необходимую для его использования, а именно внутреннее представление и допустимые действия.

Встроенные типы C# приведены в табл. 2.5. Они однозначно соответствуют стандартным классам библиотеки .NET, определенным в пространстве имен `System`. Как видно из таблицы, существуют несколько вариантов представления целых и вещественных величин. Программист выбирает тип каждой величины, используемой в программе, с учетом необходимого ему диапазона и точности представления данных.

Целые типы, а также символьный, вещественные и финансовый типы можно объединить под названием *арифметических типов.*

Внутреннее представление величины *целого типа* — целое число в двоичном коде. В знаковых типах старший бит числа интерпретируется как знаковый (0 — положительное число, 1 — отрицательное). Отрицательные числа чаще всего представляются в так называемом *дополнительном коде.* Для преобразования числа в дополнительный код все разряды числа, за исключением знакового, инвертируются, затем к числу прибавляется единица, и знаковому биту тоже присваивается единица. Беззнаковые типы позволяют представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа.

Таблица 2.5. Встроенные типы C#

Название	Ключевое слово	Тип .NET	Диапазон значений	Описание	Размер, битов
Логический тип	`bool`	`Boolean`	`true, false`		
Целые типы	`sbyte`	`SByte`	От –128 до 127	Со знаком	8
	`byte`	`Byte`	От 0 до 255	Без знака	8
	`short`	`Int16`	От –32 768 до 32 767	Со знаком	16
	`ushort`	`UInt16`	От 0 до 65 535	Без знака	16
	`int`	`Int32`	От $-2 \cdot 10^9$ до $2 \cdot 10^9$	Со знаком	32
	`uint`	`UInt32`	От 0 до $4 \cdot 10^9$	Без знака	32
	`long`	`Int64`	От -9×10^{18} до $9 \cdot 10^{18}$	Со знаком	64
	`ulong`	`UInt64`	От 0 до $18 \cdot 10^{18}$	Без знака	64
Символьный тип	`char`	`Char`	От U+0000 до U+ffff	Unicode-символ	16
Вещественные[1]	`float`	`Single`	От $1.5 \cdot 10^{-45}$ до $3.4 \cdot 10^{38}$	7 цифр	32
	`double`	`Double`	От $5.0 \cdot 10^{-324}$ до $1.7 \cdot 10^{308}$	15–16 цифр	64
Финансовый тип	`decimal`	`Decimal`	От $1.0 \cdot 10^{-28}$ до $7.9 \cdot 10^{28}$	28–29 цифр	128
Строковый тип	`string`	`String`	Длина ограничена объемом доступной памяти	Строка из Unicode-символов	
Тип object	`object`	`Object`	Можно хранить все что угодно	Всеобщий предок	

ПРИМЕЧАНИЕ

Если под величину отведено n двоичных разрядов, то в ней можно представить 2^n различных сочетаний нулей и единиц. Если старший бит отведен под знак, то диапазон возможных значений величины — $[-2^{n-1}, 2^{n-1} - 1]$, а если все разряды используются для представления значения, диапазон смещается в область положительных чисел и равен $[0, 2^n - 1]$ (см. табл. 2.5).

Вещественные типы, или *типы данных с плавающей точкой*, хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей — мантиссы и порядка, каждая часть имеет знак. Длина мантиссы определяет *точность* числа, а длина порядка — его *диапазон*. В первом приближении это можно представить себе так: например, для числа $0{,}381 \cdot 10^4$

[1] Для вещественных и финансового типов в таблице приведены абсолютные величины минимальных и максимальных значений.

хранятся цифры мантиссы 381 и порядок 4, для числа $560{,}3 \cdot 10^2$ — мантисса 5603 и порядок 5 (мантисса нормализуется), а число 0,012 представлено как 12 и 1. Конечно, в этом примере не учтены система счисления и другие особенности.

Все вещественные типы могут представлять как положительные, так и отрицательные числа. Чаще всего в программах используется тип `double`, поскольку его диапазон и точность покрывают большинство потребностей. Этот тип имеют вещественные литералы и многие стандартные математические функции.

ПРИМЕЧАНИЕ

Обратите внимание на то, что при одинаковом количестве байтов, отводимых под величины типа `float` и `int`, диапазоны их допустимых значений сильно различаются из-за внутренней формы представления. То же самое относится к `long` и `double`.

Тип `decimal` предназначен для *денежных вычислений*, в которых критичны ошибки округления. Как видно из табл. 2.5, тип `float` позволяет хранить одновременно всего 7 значащих десятичных цифр, тип `double` — 15–16. При вычислениях ошибки округления накапливаются, и при определенном сочетании значений это даже может привести к результату, в котором не будет ни одной верной значащей цифры! Величины типа `decimal` позволяют хранить 28–29 десятичных разрядов.

Тип `decimal` не относится к вещественным типам, у них различное внутреннее представление. Величины денежного типа даже нельзя использовать в одном выражении с вещественными без явного преобразования типа. Использование величин финансового типа в одном выражении с целыми допускается.

Любой встроенный тип C# соответствует стандартному классу библиотеки .NET, определенному в пространстве имен `System`. Везде, где используется имя встроенного типа, его можно заменить именем класса библиотеки. Это значит, что у встроенных типов данных C# есть методы и поля. С их помощью можно, например, получить минимальные и максимальные значения для целых, символьных, финансовых и вещественных чисел:

- `double.MaxValue` (или `System.Double.MaxValue`) — максимальное число типа `double`;
- `uint.MinValue` (или `System.UInt32.MinValue`) — минимальное число типа `uint`.

ПРИМЕЧАНИЕ

Интересно, что в вещественных классах есть элементы, представляющие положительную и отрицательную бесконечности, а также значение «не число» — это `PositiveInfinity`, `NegativeInfinity` и `NaN` соответственно. При выводе на экран, например, первого из них получится слово «бесконечность». Все доступные элементы класса можно посмотреть в окне редактора кода, введя символ точки сразу после имени типа.

Типы литералов

Как уже говорилось, величин, не имеющих типа, не существует. Поэтому литералы (константы) тоже имеют тип. Если значение целого литерала находится внутри диапазона допустимых значений типа `int`, литерал рассматривается как

int, иначе он относится к наименьшему из типов uint, long или ulong, в диапазон значений которого он входит. Вещественные литералы по умолчанию относятся к типу double.

Например, константа 10 относится к типу int (хотя для ее хранения достаточно и байта), а константа 2147483648 будет определена как uint. Для явного задания типа литерала служит суффикс, например, 1.1f, 1UL, 1000m (все суффиксы перечислены в табл. 2.3). Явное задание применяется в основном для уменьшения количества неявных преобразований типа, выполняемых компилятором.

Типы-значения и ссылочные типы

Чаще всего типы C# разделяют *по способу хранения элементов* на типы-значения и ссылочные типы (рис. 2.2)[1]. Элементы *типов-значений*, или *значимых типов* (value types), представляют собой просто последовательность битов в памяти, необходимый объем которой выделяет компилятор. Иными словами, величины значимых типов хранят свои значения непосредственно. Величина *ссылочного типа* хранит не сами данные, а ссылку на них (адрес, по которому расположены данные). Сами данные хранятся в хипе.

ВНИМАНИЕ

Несмотря на различия в способе хранения, и типы-значения, и ссылочные типы являются потомками общего базового класса object.

Рис. 2.2. Классификация типов данных C# по способу хранения

Рисунок 2.3 иллюстрирует разницу между величинами значимого и ссылочного типов. Одни и те же действия над ними выполняются по-разному. Рассмотрим в качестве примера проверку на равенство. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на

[1] Встроенные типы на рисунке выделены полужирным шрифтом, простые типы подчеркнуты.

одни и те же данные (на рисунке b и c равны, но a не равно b даже при одинаковых значениях). Из этого следует, что если изменить значение одной величины ссылочного типа, это может отразиться на другой.

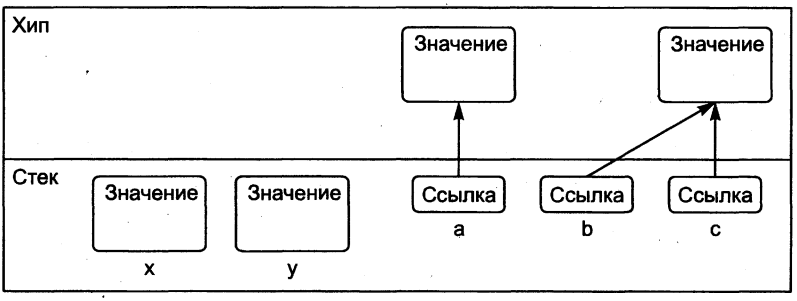


Рис. 2.3. Хранение в памяти величин значимого и ссылочного типов

Обратите внимание на то, что не все значимые типы являются простыми. По другой классификации структуры и перечисления относятся к структурированным типам, определяемым программистом. Мы рассмотрим эти типы в главе 9. Ссылочные типы мы будем изучать в главе 5 и последующих главах, после основных операторов C#, когда вы освоите синтаксис языка, а до этого ограничимся использованием встроенных типов-значений. Типы nullable введены в версию C# 2.0 и рассматриваются в главе 13.

Упаковка и распаковка

Для того чтобы величины ссылочного и значимого типов могли использоваться совместно, необходимо иметь возможность преобразования из одного типа в другой. Язык C# обеспечивает такую возможность. Преобразование из типа-значения в ссылочный тип называется *упаковкой* (boxing), обратное преобразование — *распаковкой* (unboxing).

Если величина значимого типа используется в том месте, где требуется ссылочный тип, автоматически выполняется создание промежуточной величины ссылочного типа: создается ссылка, в хипе выделяется соответствующий объем памяти и туда копируется значение величины, то есть значение как бы упаковывается в объект. При необходимости обратного преобразования с величины ссылочного типа «снимается упаковка», и в дальнейших действиях участвует только ее значение.

Рекомендации по программированию

Понятия, введенные в этой главе, являются базой для всего дальнейшего материала. На первый взгляд, изучение видов лексем может показаться излишним (пусть их различает компилятор!), однако это совершенно не так. Для того чтобы

читать программы, необходимо понимать, из каких элементов языка они состоят. Это помогает и при поиске ошибок, и при обращении к справочной системе, и при изучении новых версий языка. Более того, изучение любого нового языка рекомендуется начинать именно с лексем, которые в нем поддерживаются.

Понятие типа данных лежит в основе большинства языковых средств. При изучении любого типа необходимо рассмотреть две вещи: его внутреннее представление (а следовательно, множество возможных значений величин этого типа), а также что можно делать с этими величинами. Множество типов данных, реализуемых в языке, является одной из его важнейших характеристик. Выбор наиболее подходящего типа для представления данных — одно из необходимых условий создания эффективных программ.

Новые языки и средства программирования появляются непрерывно, поэтому программист вынужден учиться всю жизнь. Следовательно, очень важно сразу научиться учиться быстро и эффективно. Для этого надо подходить к освоению каждого языка системно: выделить составные части, понять их организацию и взаимосвязь, найти сходства и отличия от средств, изученных ранее, — короче говоря, за минимальное время разложить все в мозгу «по полочкам» так, чтобы новые знания гармонично дополнили имеющиеся. Только в этом случае ими будет легко и приятно пользоваться.

Программист-профессионал должен уметь:

- грамотно поставить задачу;
- выбрать соответствующие языковые средства;
- выбрать наиболее подходящие для представления данных структуры;
- разработать эффективный алгоритм;
- написать и документировать надежную и легко модифицируемую программу;
- обеспечить ее исчерпывающее тестирование.

Кроме того, все это необходимо выполнять в заранее заданные сроки. Надеюсь, что эта книга даст вам первоначальный импульс в нужном направлении, а также ключ к дальнейшему совершенствованию в программировании как на C#, так и на других языках.

Глава 3
Переменные, операции и выражения

В этой главе рассказывается о переменных, основных операциях C#, простейших средствах ввода-вывода и наиболее употребительных математических функциях. Приведенные сведения позволят вам создавать простые линейные программы и являются базовыми для всего дальнейшего изучения материала.

Переменные

Переменная — это именованная область памяти, предназначенная для хранения данных определенного типа. Во время выполнения программы значение переменной можно изменять. Все переменные, используемые в программе, должны быть описаны явным образом. При описании для каждой переменной задаются ее *имя* и *тип*.

Пример описания целой переменной с именем `a` и вещественной переменной `x`:

```
int a; float x;
```

Имя переменной служит для обращения к области памяти, в которой хранится *значение* переменной. Имя дает программист. Оно должно соответствовать правилам именования идентификаторов C#, отражать смысл хранимой величины и быть легко распознаваемым. Например, если в программе вычисляется количество каких-либо предметов, лучше назвать соответствующую переменную `quantity` или, на худой конец, `kolich`, но не, скажем, `A`, `t17_xz` или `prikol`.

СОВЕТ

Желательно, чтобы имя не содержало символов, которые можно перепутать друг с другом, например `l` (строчная буква `L`) и `I` (прописная буква `i`).

Тип переменной выбирается, исходя из диапазона и требуемой точности представления данных. Например, нет необходимости заводить переменную вещественного типа для хранения величины, которая может принимать только целые значения, — хотя бы потому, что целочисленные операции выполняются гораздо быстрее.

При объявлении можно присвоить переменной некоторое начальное значение, то есть *инициализировать* ее, например:

```
int a, b = 1;
float x = 0.1, y = 0.1f;
```

Здесь описаны:

- переменная `a` типа `int`, начальное значение которой не присваивается;
- переменная `b` типа `int`, ее начальное значение равно 1;
- переменные `x` и `y` типа `float`, которым присвоены одинаковые начальные значения 0.1. Разница между ними состоит в том, что для инициализации переменной `x` сначала формируется константа типа `double` (это тип, присваиваемый по умолчанию литералам с дробной частью), а затем она преобразуется к типу `float`; переменной `y` значение 0.1 присваивается без промежуточного преобразования.

При инициализации можно использовать не только константу, но и выражение — главное, чтобы на момент описания оно было вычисляемым, например:

```
int b = 1, a = 100;
int x = b * a + 25;
```

Инициализировать переменную прямо при объявлении не обязательно, но перед тем, как ее использовать в вычислениях, это сделать все равно придется, иначе компилятор сообщит об ошибке.

ВНИМАНИЕ

Рекомендуется всегда инициализировать переменные при описании.

Впрочем, иногда эту работу делает за программиста компилятор, это зависит от местонахождения описания переменной. Как вы помните из главы 1, программа на C# состоит из классов, внутри которых описывают методы и данные. Переменные, описанные непосредственно внутри класса, называются *полями класса.* Им автоматически присваивается так называемое «значение по умолчанию» — как правило, это 0 соответствующего типа.

Переменные, описанные внутри метода класса, называются *локальными переменными.* Их инициализация возлагается на программиста.

ПРИМЕЧАНИЕ

В этой главе рассматриваются только локальные переменные простых встроенных типов данных.

Так называемая *область действия* переменной, то есть область программы, где можно использовать переменную, начинается в точке ее описания и длится до конца блока, внутри которого она описана. *Блок* — это код, заключенный в фигурные скобки. Основное назначение блока — группировка операторов. В C# любая переменная описана внутри какого-либо блока: класса, метода или блока внутри метода.

Имя переменной должно быть уникальным в области ее действия. Область действия распространяется на вложенные в метод блоки, из этого следует, что во вложенном блоке нельзя объявить переменную с таким же именем, что и в охватывающем его, например:

```
class X                 // начало описания класса X
{
   int A;               // поле A класса X
   int B;               // поле B класса X

   void Y()             // ---------------------------------------- метод Y класса X
   {
      int C;            // локальная переменная C, область действия - метод Y
      int A;            // локальная переменная A (НЕ конфликтует с полем A)

      {                 // ============ вложенный блок 1 ============
         int D;         // локальная переменная D, область действия - этот блок
//       int A;            недопустимо! Ошибка компиляции, конфликт с локальной
//                         переменной A
         C = B;         // присваивание переменной C поля B класса X (**)
         C = this.A;    // присваивание переменной C поля A класса X (***)
      }                 // ============ конец блока 1 ===============

      {                 // ============ вложенный блок 2 ============
         int D;         // локальная переменная D, область действия - этот блок
      }                 // ============ конец блока 2 ===============

   }                    // ------------------------ конец описания метода Y класса X

}                       // конец описания класса X
```

Давайте разберемся в этом примере. В нем описан класс `X`, содержащий три элемента: поле `A`, поле `B` и метод `Y`. Непосредственно внутри метода `Y` заданы две локальные переменные — `C` и `A`.

Внутри метода класса можно описывать переменную с именем, совпадающим с полем класса, потому что существует способ доступа к полю класса с помощью ключевого слова `this` (это иллюстрирует строка, отмеченная символами ***). Таким образом, локальная переменная `A` не «закрывает» поле `A` класса `X`, а вот попытка описать во вложенном блоке другую локальную переменную с тем же именем окончится неудачей (эти строки закомментированы).

Если внутри метода нет локальных переменных, совпадающих с полями класса, к этим полям можно обратиться в методе непосредственно (см. строку, помеченную

символами **). Две переменные с именем D не конфликтуют между собой, поскольку блоки, в которых они описаны, не вложены один в другой.

СОВЕТ

Как правило, переменным с большой областью действия даются более длинные имена, а для переменных, вся «жизнь» которых — несколько строк исходного текста, хватит и одной буквы с комментарием при объявлении.

В листинге 3.1 приведен пример программы, в которой описываются и выводятся на экран локальные переменные.

Листинг 3.1. Описание переменных

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            int     i = 3;
            double  y = 4.12;
            decimal d = 600m;
            string  s = "Вася";

            Console.Write( "i = " );  Console.WriteLine( i );
            Console.Write( "y = " );  Console.WriteLine( y );
            Console.Write( "d = " );  Console.WriteLine( d );
            Console.Write( "s = " );  Console.WriteLine( s );
        }
    }
}
```

Как вы догадались, метод Write делает то же самое, что и WriteLine, но не переводит строку. Более удобные способы вывода рассмотрены в конце этой главы в разделе «Простейший ввод-вывод» (см. с. 59).

ВНИМАНИЕ

Переменные создаются при входе в их область действия (блок) и уничтожаются при выходе. Это означает, что после выхода из блока значение переменной не сохраняется. При повторном входе в этот же блок переменная создается заново.

Именованные константы

Можно запретить изменять значение переменной, задав при ее описании ключевое слово const, например:

```
const int b = 1;
const float x = 0.1, y = 0.1f;   // const распространяется на обе переменные
```

Такие величины называют *именованными константами*, или просто *константами*. Они применяются для того, чтобы вместо значений констант можно было использовать в программе их имена. Это делает программу более понятной и облегчает внесение в нее изменений, поскольку изменить значение достаточно только в одном месте программы.

ПРИМЕЧАНИЕ

Улучшение читабельности происходит только при осмысленном выборе имен констант. В хорошо написанной программе вообще не должно встречаться иных чисел, кроме 0 и 1, все остальные числа должны задаваться именованными константами с именами, отражающими их назначение.

Именованные константы должны обязательно инициализироваться при описании. При инициализации можно использовать не только константу, но и выражение — главное, чтобы оно было вычисляемым на этапе компиляции, например:

```
const int b = 1, a = 100;
const int x = b * a + 25;
```

Операции и выражения

Выражение — это правило вычисления значения. В выражении участвуют *операнды*, объединенные *знаками операций*. Операндами простейшего выражения могут быть константы, переменные и вызовы функций.

Например, `a + 2` — это выражение, в котором + является знаком операции, а `a` и `2` — операндами. Пробелы внутри знака операции, состоящей из нескольких символов, не допускаются. По количеству участвующих в одной операции операндов операции делятся на *унарные*, *бинарные* и *тернарную*. Операции C# приведены в табл. 3.1[1].

Таблица 3.1. Операции C#

Категория	Знак операции	Название	Описание
Первичные	`.`	Доступ к элементу	С. 105
	`x()`	Вызов метода или делегата	С. 108, 221
	`x[]`	Доступ к элементу	С. 127
	`x++`	Постфиксный инкремент	С. 47
	`x--`	Постфиксный декремент	С. 47
	`new`	Выделение памяти	С. 48
	`typeof`	Получение типа	С. 280
	`checked`	Проверяемый код	С. 46
	`unchecked`	Непроверяемый код	С. 46

[1] В этой таблице символ *x* призван показать расположение операнда и не является частью знака операции.

Категория	Знак операции	Название	Описание
Унарные	+	Унарный плюс	
	-	Унарный минус (арифметическое отрицание)	С. 48
	!	Логическое отрицание	С. 48
	~	Поразрядное отрицание	С. 48
	++x	Префиксный инкремент	С. 47
	--x	Префиксный декремент	С. 47
	(тип)x	Преобразование типа	С. 49
Мультипликативные (типа умножения)	*	Умножение	С. 50
	/	Деление	С. 50
	%	Остаток от деления	С. 50
Аддитивные (типа сложения)	+	Сложение	С. 43
	-	Вычитание	С. 53
Сдвига	<<	Сдвиг влево	С. 54
	>>	Сдвиг вправо	С. 54
Отношения и проверки типа	<	Меньше	С. 54
	>	Больше	С. 54
	<=	Меньше или равно	С. 54
	>=	Больше или равно	С. 54
	is	Проверка принадлежности типу	С. 194
	as	Приведение типа	С. 194
Проверки на равенство	==	Равно	С. 54
	!=	Не равно	С. 54
Поразрядные логические	&	Поразрядная конъюнкция (И)	С. 55
	^	Поразрядное исключающее ИЛИ	С. 55
	|	Поразрядная дизъюнкция (ИЛИ)	С. 55
Условные логические	&&	Логическое И	С. 56
	||	Логическое ИЛИ	С. 56
Условная	? :	Условная операция	С. 56
Присваивания	=	Присваивание	С. 56
	*=	Умножение с присваиванием	
	/=	Деление с присваиванием	
	%=	Остаток отделения с присваиванием	
	+=	Сложение с присваиванием	

продолжение

Таблица 3.1 (*продолжение*)

Категория	Знак операции	Название	Описание
	-=	Вычитание с присваиванием	
	<<=	Сдвиг влево с присваиванием	
	>>=	Сдвиг вправо с присваиванием	
	&=	Поразрядное И с присваиванием	
	^=	Поразрядное исключающее ИЛИ с присваиванием	
	|=	Поразрядное ИЛИ с присваиванием	

ПРИМЕЧАНИЕ

В версию C# 2.0 введена операция объединения ??, которая рассматривается в главе 13 (см. раздел «Обнуляемые типы», с. 309).

Операции в выражении выполняются в определенном порядке в соответствии с *приоритетами*, как и в математике. В табл. 3.1 операции расположены по убыванию приоритетов, уровни приоритетов разделены в таблице горизонтальными линиями.

Результат вычисления выражения характеризуется *значением* и *типом*. Например, пусть a и b — переменные целого типа и описаны так:

```
int a = 2, b = 5;
```

Тогда выражение a + b имеет значение 7 и тип int, а выражение a = b имеет значение, равное помещенному в переменную a (в данному случае — 5), и тип, совпадающий с типом этой переменной.

Если в одном выражении соседствуют несколько операций одинакового приоритета, операции присваивания и условная операция выполняются *справа налево*, остальные — *слева направо*. Для изменения порядка выполнения операций используются *круглые скобки*, уровень их вложенности практически не ограничен.

Например, a + b + c означает (a + b) + c, а a = b = c означает a = (b = c). То есть сначала вычисляется выражение b = c, а затем его результат становится правым операндом для операции присваивания переменной a.

ПРИМЕЧАНИЕ

Часто перед выполнением операции требуется вычислить значения операндов. Например, в выражении F(i) + G(i++) * H(i) сначала вызываются функции F, G и H, а затем выполняются умножение и сложение. Операнды всегда вычисляются слева направо независимо от приоритетов операций, в которых они участвуют. Кстати, в приведенном примере метод H вызывается с новым значением i (увеличенным на 1).

Тип результата выражения в общем случае формируется по правилам, которые описаны в следующем разделе.

Преобразования встроенных арифметических типов-значений

При вычислении выражений может возникнуть необходимость в преобразовании типов. *Если операнды, входящие в выражение, одного типа и операция для этого типа определена, то результат выражения будет иметь тот же тип.*

Если операнды разного типа и/или операция для этого типа не определена, перед вычислениями автоматически выполняется преобразование типа по правилам, обеспечивающим приведение более коротких типов к более длинным для сохранения значимости и точности. Автоматическое (*неявное*) преобразование возможно не всегда, а только если при этом не может случиться потеря значимости.

Если неявного преобразования из одного типа в другой не существует, программист может задать *явное* преобразование типа с помощью операции `(тип)x`. Его результат остается на совести программиста. Явное преобразование рассматривается в этой главе немного позже.

ВНИМАНИЕ

Арифметические операции не определены для более коротких, чем `int`, типов. Это означает, что если в выражении участвуют только величины типов `sbyte`, `byte`, `short` и `ushort`, перед выполнением операции они будут преобразованы в `int`. Таким образом, результат любой арифметической операции имеет тип не менее `int`.

Правила неявного преобразования иллюстрирует рис. 3.1. Если один из операндов имеет тип, изображенный на более низком уровне, чем другой, то он приводится к типу второго операнда при наличии пути между ними. Если пути нет, возникает ошибка компиляции. Если путей несколько, выбирается наиболее короткий, не содержащий пунктирных линий. Преобразование выполняется не последовательно, а непосредственно из исходного типа в результирующий.

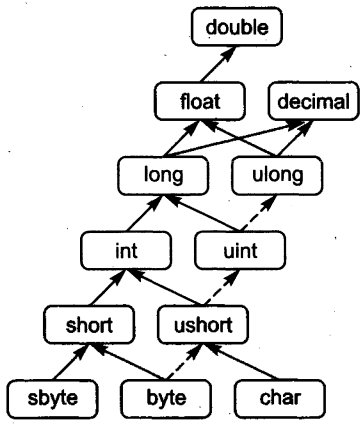


Рис. 3.1. Неявные арифметические преобразования типов

Преобразование более коротких, чем int, типов выполняется при присваивании. Обратите внимание на то, что неявного преобразования из float и double в decimal не существует.

ПРИМЕЧАНИЕ

Преобразование из типов int, uint и long в тип float и из типа long в тип double может вызвать потерю точности, но не потерю значимости. В процессе других вариантов неявного преобразования никакая информация не теряется.

Введение в исключения

При вычислении выражений могут возникнуть ошибки, например, переполнение, исчезновение порядка или деление на ноль. В C# есть механизм, который позволяет обрабатывать подобные ошибки и таким образом избегать аварийного завершения программы. Он так и называется: *механизм обработки исключительных ситуаций* (*исключений*).

Если в процессе вычислений возникла ошибка, система сигнализирует об этом с помощью специального действия, называемого *выбрасыванием* (*генерированием*) *исключения*. Каждому типу ошибки соответствует свое исключение. Поскольку C# — язык объектно-ориентированный, исключения являются классами, которые имеют общего предка — класс Exception, определенный в пространстве имен System.

Например, при делении на ноль будет выброшено (сгенерировано) исключение с длинным, но понятным именем DivideByZeroException, при недостатке памяти — исключение OutOfMemoryException, при переполнении — исключение OverflowException.

ПРИМЕЧАНИЕ

Стандартных исключений очень много, тем не менее программист может создавать и собственные исключения на основе класса Exception.

Программист может задать способ обработки исключения в специальном блоке кода, начинающемся с ключевого слова catch («перехватить»), который будет автоматически выполнен при возникновении соответствующей исключительной ситуации. Внутри блока можно, например, вывести предупреждающее сообщение или скорректировать значения величин и продолжить выполнение программы. Если этот блок не задан, система выполнит *действия по умолчанию*, которые обычно заключаются в выводе диагностического сообщения и нормальном завершении программы.

Процессом выбрасывания исключений, возникающих при переполнении, можно управлять. Для этого служат ключевые слова checked и unchecked. Слово checked включает проверку переполнения, слово unchecked выключает. При выключенной проверке исключения, связанные с переполнением, не генерируются, а результат операции усекается. Проверку переполнения можно реализовать для отдельного выражения или для целого блока операторов, например:

```
a = checked (b + c);          // для выражения
unchecked {                   // для блока операторов
   a = b + c;
}
```

Проверка не распространяется на функции, вызванные в блоке. Если проверка переполнения включена, говорят, что вычисления выполняются *в проверяемом контексте*, если выключена — в непроверяемом. Проверку переполнения выключают в случаях, когда усечение результата операции необходимо в соответствии с алгоритмом.

Можно задать проверку переполнения во всей программе с помощью ключа компилятора /checked, это полезно при отладке программы. Поскольку подобная проверка несколько замедляет работу, в готовой программе этот режим обычно не используется.

Мы подробно рассмотрим исключения и их обработку в разделе «Обработка исключительных ситуаций» (см. с. 89).

Основные операции C#

В этом разделе кратко описаны синтаксис и применение всех операций C#, кроме некоторых первичных, которые рассматриваются в последующих главах при изучении соответствующего материала.

Инкремент и декремент

Операции инкремента (++) и декремента (--), называемые также операциями увеличения и уменьшения на единицу, имеют две формы записи — *префиксную*, когда знак операции записывается перед операндом, и *постфиксную*. В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется. Листинг 3.2 иллюстрирует эти операции.

Листинг 3.2. Операции инкремента и декремента

```
using System;
namespace ConsoleApplication1
{    class Class1
    {    static void Main()
        {
            int x = 3, y = 3;
            Console.Write( "Значение префиксного выражения: " );
            Console.WriteLine( ++x );
            Console.Write( "Значение x после приращения: " );
            Console.WriteLine( x );

            Console.Write( "Значение постфиксного выражения: " );
            Console.WriteLine( y++ );
            Console.Write( "Значение y после приращения: " );
            Console.WriteLine( y );
        }
    }
}
```

Результат работы программы:

```
Значение префиксного выражения: 4
Значение x после приращения: 4
Значение постфиксного выражения: 3
Значение y после приращения: 4
```

Стандартные операции инкремента существуют для целых, символьных, вещественных и финансовых величин, а также для перечислений. Операндом может быть переменная, свойство или индексатор (мы рассмотрим свойства и индексаторы в свое время, в главах 5 и 7).

Операция new

Операция new служит для создания нового объекта. Формат операции:

```
new тип ( [ аргументы ] )
```

С помощью этой операции можно создавать объекты как ссылочных, так и значимых типов, например:

```
object z = new object();
int i = new int();          // то же самое, что int i = 0;
```

Объекты ссылочного типа обычно формируют именно этим способом, а переменные значимого типа чаще создаются так, как описано ранее в разделе «Переменные».

При выполнении операции new сначала выделяется необходимый объем памяти (для ссылочных типов в хипе, для значимых — в стеке), а затем вызывается так называемый *конструктор по умолчанию*, то есть метод, с помощью которого инициализируется объект. Переменной значимого типа присваивается *значение по умолчанию*, которое равно нулю соответствующего типа. Для ссылочных типов стандартный конструктор инициализирует значениями по умолчанию все поля объекта.

Если необходимый для хранения объекта объем памяти выделить не удалось, генерируется исключение OutOfMemoryException.

Операции отрицания

Арифметическое отрицание (унарный минус -) меняет знак операнда на противоположный. Стандартная операция отрицания определена для типов int, long, float, double и decimal. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам (см. рис. 3.1). Тип результата соответствует типу операции.

ПРИМЕЧАНИЕ

Для значений целого и финансового типов результат достигается вычитанием исходного значения из нуля. При этом может возникнуть переполнение. Будет ли при этом выброшено исключение, зависит от контекста.

Логическое отрицание (!) определено для типа bool. Результат операции — значение false, если операнд равен true, и значение true, если операнд равен false.

Поразрядное отрицание (~), часто называемое побитовым, инвертирует каждый разряд в двоичном представлении операнда типа int, uint, long или ulong.

Операции отрицания представлены в листинге 3.3.

Листинг 3.3. Операции отрицания

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            sbyte a = 3, b = -63, c = 126;
            bool  d = true;
            Console.WriteLine( -a );   // Результат -3
            Console.WriteLine( -c );   // Результат -126
            Console.WriteLine( !d );   // Результат false
            Console.WriteLine( ~a );   // Результат -4
            Console.WriteLine( ~b );   // Результат 62
            Console.WriteLine( ~c );   // Результат -127
        }
    }
}
```

Явное преобразование типа

Операция используется, как и следует из ее названия, для явного преобразования величины из одного типа в другой. Это требуется в том случае, когда неявного преобразования не существует. При преобразовании из более длинного типа в более короткий возможна потеря информации, если исходное значение выходит за пределы диапазона результирующего типа[1].

Формат операции:

(тип) выражение

Здесь тип — это имя того типа, в который осуществляется преобразование, а выражение чаще всего представляет собой имя переменной, например:

```
long b = 300;
int  a = (int) b;       // данные не теряются
byte d = (byte) a;      // данные теряются
```

Преобразование типа часто применяется для ссылочных типов при работе с иерархиями объектов.

[1] Эта потеря никак не диагностируется, то есть остается на совести программиста.

Умножение, деление и остаток от деления

Операция умножения (*) возвращает результат перемножения двух операндов. Стандартная операция умножения определена для типов `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам (см. рис. 3.1). Тип результата операции равен «наибольшему» из типов операндов, но не менее `int`.

Если оба операнда целочисленные или типа `decimal` и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение `System.OverflowException`[1].

Все возможные значения для вещественных операндов приведены в табл. 3.2. Символами `x` и `y` обозначены конечные положительные значения, символом `z` — результат операции вещественного умножения. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность»[2], если слишком мал, он принимается за 0. `NaN` (not a number) означает, что результат не является числом.

Таблица 3.2. Результаты вещественного умножения

*	+y	−y	+0	−0	+∞	−∞	NaN
+x	+z	−z	+0	−0	+∞	−∞	NaN
−x	−z	+z	−0	+0	−∞	+∞	NaN
+0	+0	−0	+0	−0	NaN	NaN	NaN
−0	−0	+0	−0	+0	NaN	NaN	NaN
+∞	+∞	−∞	NaN	NaN	+∞	−∞	NaN
−∞	−∞	+∞	NaN	NaN	−∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Операция деления (/) вычисляет частное от деления первого операнда на второй. Стандартная операция деления определена для типов `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`. К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам. Тип результата определяется правилами преобразования (см. рис. 3.1), но не меньше `int`.

Если оба операнда целочисленные, результат операции округляется вниз до ближайшего целого числа. Если делитель равен нулю, генерируется исключение `System.DivideByZeroException`.

Если хотя бы один из операндов вещественный, дробная часть результата деления не отбрасывается, а все возможные значения приведены в табл. 3.3. Символами `x` и `y` обозначены конечные положительные значения, символом `z` — результат

[1] В проверяемом контексте. В непроверяемом исключение не выбрасывается, зато отбрасываются избыточные биты.

[2] Об «особых» значениях вещественных величин упоминалось на с. 34.

операции вещественного деления. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность», если слишком мал, он принимается за 0.

Таблица 3.3. Результаты вещественного деления

/	+y	−y	+0	−0	+∞	−∞	NaN
+x	+z	−z	+∞	−∞	+0	−0	NaN
−x	−z	+z	−∞	+∞	−0	+0	NaN
+0	+0	−0	NaN	NaN	+0	−0	NaN
−0	−0	+0	NaN	NaN	−0	+0	NaN
+∞	+∞	−∞	+∞	−∞	NaN	NaN	NaN
−∞	−∞	+∞	−∞	+∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Для финансовых величин (тип `decimal`) при делении на 0 и переполнении генерируются соответствующие исключения, при исчезновении порядка результат равен 0.

Операция остатка от деления (`%`) также интерпретируется по-разному для целых, вещественных и финансовых величин. Если оба операнда целочисленные, результат операции вычисляется по формуле `x - (x / y) * y`. Если делитель равен нулю, генерируется исключение `System.DivideByZeroException`. Тип результата операции равен «наибольшему» из типов операндов, но не менее `int` (см. рис. 3.1).

Если хотя бы один из операндов вещественный, результат операции вычисляется по формуле `x - n * y`, где `n` — наибольшее целое, меньшее или равное результату деления `x` на `y`. Все возможные комбинации значений операндов приведены в табл. 3.4. Символами `x` и `y` обозначены конечные положительные значения, символом `z` — результат операции остатка от деления.

Таблица 3.4. Результаты вещественного остатка от деления

%	+y	−y	+0	−0	+∞	−∞	NaN
+x	+z	z	NaN	NaN	x	x	NaN
−x	−z	−z	NaN	NaN	−x	−x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
−0	−0	−0	NaN	NaN	−0	−0	NaN
+∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
−∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Для финансовых величин (тип `decimal`) при получении остатка от деления на 0 и при переполнении генерируются соответствующие исключения, при исчезновении порядка результат равен 0. Знак результата равен знаку первого операнда.

Пример применения операциий умножения, деления и получения остатка представлен в листинге 3.4.

Листинг 3.4. Операции умножения, деления и получения остатка

```
using System;
namespace ConsoleApplication1
{ class Class1
   { static void Main()
      {
         int x = 11, y = 4;
         float z = 4;
         Console.WriteLine( z * y );            // Результат 16
         Console.WriteLine( z * 1e308 );        // Результат "бесконечность"
         Console.WriteLine( x / y );            // Результат 2
         Console.WriteLine( x / z );            // Результат 2,75
         Console.WriteLine( x % y );            // Результат 3
         Console.WriteLine( 1e-324 / 1e-324 ); // Результат NaN
      }
   }
}
```

Еще раз обращаю ваше внимание на то, что несколько операций одного приоритета выполняются слева направо. Для примера рассмотрим выражение $2 / x \cdot y$. Деление и умножение имеют один и тот же приоритет, поэтому сначала 2 делится на x, а затем результат этих вычислений умножается на y. Иными словами, это выражение эквивалентно формуле

$$\frac{2}{x} \cdot y$$

Если же мы хотим, чтобы выражение $x \cdot y$ было в знаменателе, следует заключить его в круглые скобки или сначала поделить числитель на x, а потом на y, то есть записать как $2 / (x * y)$ *или* $2 / x / y$.

Сложение и вычитание

Операция сложения (+) возвращает сумму двух операндов. Стандартная операция сложения определена для типов `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`. К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам (см. рис. 3.1). Тип результата операции равен «наибольшему» из типов операндов, но не менее `int`.

Если оба операнда целочисленные или типа `decimal` и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение `System.OverflowException`[1].

[1] В проверяемом контексте. В непроверяемом исключение не выбрасывается, зато отбрасываются избыточные биты.

Все возможные значения для вещественных операндов приведены в табл. 3.5. Символами x и y обозначены конечные положительные значения, символом z — результат операции вещественного сложения. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность», если слишком мал, он принимается за 0.

Таблица 3.5. Результаты вещественного сложения

+	y	+0	−0	+∞	−∞	NaN
x	z	x	x	+∞	−∞	NaN
+0	y	+0	+0	+∞	−∞	NaN
−0	y	+0	−0	+∞	−∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
−∞	−∞	−∞	−∞	NaN	−∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

Операция вычитания (-) возвращает разность двух операндов. Стандартная операция вычитания определена для типов `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`. К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам (см. рис. 3.1). Тип результата операции равен «наибольшему» из типов операндов, но не менее `int`.

Если оба операнда целочисленные или типа `decimal` и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение `System.OverflowException`.

Все возможные значения результата вычитания для вещественных операндов приведены в табл. 3.6. Символами x и y обозначены конечные положительные значения, символом z — результат операции вещественного вычитания. Если x и y равны, результат равен положительному нулю. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность» с тем же знаком, что и x - y, если слишком мал, он принимается за 0 с тем же знаком, что и x - y.

Таблица 3.6. Результаты вещественного вычитания

−	y	+0	−0	+∞	−∞	NaN
x	z	x	x	−∞	+∞	NaN
+0	−y	+0	+0	−∞	+∞	NaN
−0	−y	−0	+0	−∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
−∞	−∞	−∞	−∞	−∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

Операции сдвига

Операции сдвига (<< и >>) применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом[1].

При *сдвиге влево* (<<) освободившиеся разряды обнуляются. При *сдвиге вправо* (>>) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа (то есть выполняется логический сдвиг), и знаковым разрядом — в противном случае (выполняется арифметический сдвиг). Операции сдвига никогда не приводят к переполнению и потере значимости. Стандартные операции сдвига определены для типов `int`, `uint`, `long` и `ulong`.

Пример применения операций сдвига представлен в листинге 3.5.

Листинг 3.5. Операции сдвига

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            byte  a = 3, b = 9;
            sbyte c = 9, d = -9;
            Console.WriteLine( a << 1 );        // Результат 6
            Console.WriteLine( a << 2 );        // Результат 12
            Console.WriteLine( b >> 1 );        // Результат 4
            Console.WriteLine( c >> 1 );        // Результат 4
            Console.WriteLine( d >> 1 );        // Результат -5
        }
    }
}
```

Операции отношения и проверки на равенство

Операции отношения (<, <=, >, >=, ==, !=) сравнивают первый операнд со вторым. Операнды должны быть арифметического типа. Результат операции — логического типа, равен `true` или `false`. Правила вычисления результатов приведены в табл. 3.7.

Таблица 3.7. Результаты операций отношения

Операция	Результат
`x == y`	`true`, если `x` равно `y`, иначе `false`
`x != y`	`true`, если `x` не равно `y`, иначе `false`
`x < y`	`true`, если `x` меньше `y`, иначе `false`
`x > y`	`true`, если `x` больше `y`, иначе `false`
`x <= y`	`true`, если `x` меньше или равно `y`, иначе `false`
`x >= y`	`true`, если `x` больше или равно `y`, иначе `false`

[1] Фактически, учитывается только 5 младших битов второго операнда, если первый имеет тип `int` или `uint`, и 6 битов, если первый операнд имеет тип `long` или `ulong`.

ПРИМЕЧАНИЕ

Обратите внимание на то, что операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции сравнения.

Очень интересно формируется результат операций отношения для особых случаев вещественных значений. Например, если один из операндов равен `NaN`, результатом для всех операций, кроме `!=`, будет `false` (для операции `!=` результат равен `true`).

Очевиден факт, что для любых операндов результат операции `x != y` всегда равен результату операции `!(x == y)`, однако если один или оба операнда равны `NaN`, для операций `<`, `>`, `<=` и `>=` этот факт не подтверждается. Например, если `x` или `y` равны `NaN`, то `x < y` даст `false`, а `!(x >= y)` — `true`.

Другие особые случаи рассматриваются следующим образом:

- значения +0 и −0 равны;
- значение $-\infty$ меньше любого конечного значения и равно другому значению $-\infty$;
- значение $+\infty$ больше любого конечного значения и равно другому значению $+\infty$.

Поразрядные логические операции

Поразрядные логические операции (`&`, `|`, `^`) применяются к целочисленным операндам и работают с их двоичными представлениями. При выполнении операций операнды сопоставляются побитно (первый бит первого операнда с первым битом второго, второй бит первого операнда со вторым битом второго и т. д.). Стандартные операции определены для типов `int`, `uint`, `long` и `ulong`.

При *поразрядной конъюнкции, или поразрядном И* (операция обозначается `&`), бит результата равен 1 только тогда, когда соответствующие биты обоих операндов равны 1.

При *поразрядной дизъюнкции, или поразрядном ИЛИ* (операция обозначается `|`), бит результата равен 1 тогда, когда соответствующий бит хотя бы одного из операндов равен 1.

При поразрядном исключающем ИЛИ (операция обозначается `^`) бит результата равен 1 только тогда, когда соответствующий бит только одного из операндов равен 1.

Пример применения поразрядных логических операций представлен в листинге 3.6.

Листинг 3.6. Поразрядные логические операции

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            Console.WriteLine( 6 & 5 );        // Результат 4
            Console.WriteLine( 6 | 5 );        // Результат 7
            Console.WriteLine( 6 ^ 5 );        // Результат 3
        }
    }
}
```

Условные логические операции

Условные логические операции И (&&) и ИЛИ (||) чаще всего используются с операндами логического типа. Результатом логической операции является true или false. Операции вычисляются по сокращенной схеме.

Результат операции *логическое И* имеет значение true, только если оба операнда имеют значение true. Результат операции *логическое ИЛИ* имеет значение true, если хотя бы один из операндов имеет значение true.

ВНИМАНИЕ

Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется. Например, если первый операнд операции И равен false, результатом операции будет false независимо от значения второго операнда, поэтому он не вычисляется.

Пример применения условных логических операций представлен в листинге 3.7.

Листинг 3.7. Условные логические операции

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            Console.WriteLine( true && true  );      // Результат true
            Console.WriteLine( true && false );      // Результат false
            Console.WriteLine( true || true  );      // Результат true
            Console.WriteLine( true || false );      // Результат true
        }
    }
}
```

Условная операция

Условная операция (? :) — тернарная, то есть имеет три операнда. Ее формат:

```
операнд_1 ? операнд_2 : операнд_3
```

Первый операнд — выражение, для которого существует неявное преобразование к логическому типу. Если результат вычисления первого операнда равен true, то результатом условной операции будет значение второго операнда, иначе — третьего операнда. Вычисляется всегда либо второй операнд, либо третий. Их тип может различаться.

Тип результата операции зависит от типа второго и третьего операндов:

- если операнды одного типа, он и становится типом результата операции[1];
- иначе, если существует неявное преобразование типа от операнда 2 к операнду 3, но не наоборот, то типом результата операции становится тип операнда 3;

[1] Это наиболее часто используемый вариант применения тернарной операции.

- иначе, если существует неявное преобразование типа от операнда 3 к операнду 2, но не наоборот, то типом результата операции становится тип операнда 2;
- иначе возникает ошибка компиляции.

Условную операцию часто используют вместо условного оператора `if` (он рассматривается в следующей главе) для сокращения текста программы.

Пример применения условной операции представлен в листинге 3.8.

Листинг 3.8. Условная операция

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            int a = 11, b = 4;
            int max = b > a ? b : a;
            Console.WriteLine( max );      // Результат 11
        }
    }
}
```

Другой пример применения условной операции: требуется, чтобы некоторая целая величина увеличивалась на 1, если ее значение не превышает `n`, а иначе принимала значение 1. Это удобно реализовать следующим образом:

```
i = (i < n) ? i + 1: 1;
```

Условная операция правоассоциативна, то есть выполняется справа налево. Например, выражение `a ? b : c ? d : e` вычисляется как `a ? b : (c ? d : e)`[1].

Операции присваивания

Операции присваивания (=, +=, -=, *= и т. д.) задают новое значение переменной[2]. Эти операции могут использоваться в программе как законченные операторы.

Формат операции *простого присваивания* (=):

```
переменная = выражение
```

Механизм выполнения операции присваивания такой: вычисляется выражение и его результат заносится в память по адресу, который определяется именем переменной, находящейся слева от знака операции. То, что ранее хранилось в этой области памяти, естественно, теряется. Схематично это полезно представить себе так:

```
Переменная ← Выражение
```

Напомню, что константа и переменная являются частными случаями выражения.

---

[1] Строго говоря, такой «хитрой» записи следует избегать. Программа не должна напоминать шараду.

[2] А также свойству, событию или индексатору, которые мы рассмотрим в свое время.

Примеры операторов присваивания:

```
a = b + c / 2;
b = a;
a = b;
x = 1;
x = x + 0.5;
```

Обратите внимание: `b = a` и `a = b` — это совершенно разные действия!

ПРИМЕЧАНИЕ

Чтобы не перепутать, что чему присваивается, запомните мнемоническое правило: присваивание — это передача данных «налево».

Начинающие часто делают ошибку, воспринимая присваивание как аналог равенства в математике. Чтобы избежать этой ошибки, надо понимать механизм работы оператора присваивания. Рассмотрим для этого последний пример (`x = x + 0.5`). Сначала из ячейки памяти, в которой хранится значение переменной `x`, выбирается это значение. Затем к нему прибавляется `0.5`, после чего получившийся результат записывается в ту же самую ячейку, а то, что хранилось там ранее, теряется безвозвратно. Операторы такого вида применяются в программировании очень широко.

Для правого операнда операции присваивания должно существовать неявное преобразование к типу левого операнда. Например, выражение целого типа можно присвоить вещественной переменной, потому что целые числа являются подмножеством вещественных, и информация при таком присваивании не теряется:

```
вещественная_переменная = целое_выражение;
```

Правила преобразований перечислены в разделе «Преобразования встроенных арифметических типов-значений» (см. с. 45).

Результатом операции присваивания является значение, записанное в левый операнд. Тип результата совпадает с типом левого операнда.

В сложных операциях присваивания (`+=`, `*=`, `/=` и т. п.) при вычислении выражения, стоящего в правой части, используется значение из левой части. Например, при *сложении с присваиванием* ко второму операнду прибавляется первый, и результат записывается в первый операнд, то есть выражение `a += b` является более компактной записью выражения `a = a + b`.

Результатом операции сложного присваивания является значение, записанное в левый операнд.

ПРИМЕЧАНИЕ

В документации написано, что тип результата совпадает с типом левого операнда, если он не менее `int`. Это означает, что если, например, переменные `a` и `b` имеют тип `byte`, присваивание `a += b` недопустимо и требуется преобразовать тип явным образом: `a += (byte)b`. Однако на практике компилятор ошибку не выдает.

Напомню, что операции присваивания *правоассоциативны*, то есть выполняются справа налево, в отличие от большинства других операций (`a = b = c` означает `a = (b = c)`).

Не рассмотренные в этом разделе операции будут описаны позже.

Линейные программы

Линейной называется программа, все операторы которой выполняются последовательно в том порядке, в котором они записаны. Простейшим примером линейной программы является программа расчета по заданной формуле. Она состоит из трех этапов: ввод исходных данных, вычисление по формуле и вывод результатов. Для того чтобы написать подобную программу, нам пока не хватает знаний о том, как организовать ввод и вывод на языке C#. Подробно этот вопрос рассматривается в главе 11, а здесь приводятся только минимально необходимые сведения.

Простейший ввод-вывод

Любая программа при вводе исходных данных и выводе результатов взаимодействует с внешними устройствами. Совокупность стандартных устройств ввода и вывода, то есть клавиатуры и экрана, называется *консолью*. Обмен данными с консолью является частным случаем обмена с внешними устройствами, который подробно рассмотрен в главе 11.

В языке C#, как и во многих других, нет операторов ввода и вывода. Вместо них для обмена с внешними устройствами применяются стандартные объекты. Для работы с консолью в C# применяется класс `Console`, определенный в пространстве имен `System`. Методы этого класса `Write` и `WriteLine` уже использовались в наших программах. Поговорим о них подробнее, для чего внесем некоторые изменения в листинг 3.1. Результаты этих изменений представлены в листинге 3.9.

Листинг 3.9. Методы вывода

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            int     i = 3;
            double  y = 4.12;
            decimal d = 600m;
            string  s = "Вася";

            Console.WriteLine( "i = " + i );                    // 1
            Console.WriteLine( "y = {0} \nd = {1}", y, d );     // 2
            Console.WriteLine( "s = " + s );                    // 3
        }
    }
}
```

Результат работы программы:

```
i = 3
y = 4,12
d = 600
s = Вася
```

До сих пор мы использовали метод `WriteLine` для вывода значений переменных и литералов различных встроенных типов. Это возможно благодаря тому, что в классе `Console` существует несколько вариантов методов с именами `Write` и `WriteLine`, предназначенных для вывода значений различных типов.

Методы с одинаковыми именами, но разными параметрами называются *перегруженными*. Компилятор определяет, какой из методов вызван, по типу передаваемых в него величин. Методы вывода в классе `Console` перегружены для всех встроенных типов данных, кроме того, предусмотрены варианты форматного вывода.

Листинг 3.9 содержит два наиболее употребительных варианта вызова методов вывода. Сначала обратите внимание на способ вывода пояснений к значениям переменных в строках 1 и 3. Пояснения представляют собой строковые литералы. Если метод `WriteLine` вызван с *одним* параметром, он может быть любого встроенного типа, например, числом, символом или строкой. Нам же требуется вывести в каждой строке не одну, а две величины: текстовое пояснение и значение переменной, — поэтому прежде чем передавать их для вывода, их требуется «склеить» в одну строку с помощью операции `+`.

Перед объединением строки с числом надо преобразовать число из его внутренней формы представления в последовательность символов, то есть в строку. *Преобразование в строку* определено во всех стандартных классах C# — для этого служит метод `ToString()`. В данном случае он выполняется неявно, но можно вызвать его и явным образом:

```
Console.WriteLine( "i = " + i.ToString() );
```

Оператор 2 иллюстрирует форматный вывод. В этом случае используется другой вариант метода `WriteLine`, который содержит более одного параметра. Первым параметром методу передается строковый литерал, содержащий помимо обычных символов, предназначенных для вывода на консоль, *параметры* в фигурных скобках, а также *управляющие последовательности* (они были рассмотрены в главе 2).

Параметры нумеруются с нуля, перед выводом они заменяются значениями соответствующих переменных в списке вывода: нулевой параметр заменяется значением первой переменной (в данном примере — `y`), первый параметр — второй переменной (в данном примере — `d`) и т. д.

ПРИМЕЧАНИЕ

Для каждого параметра можно задать ширину поля вывода и формат вывода. Мы рассмотрим эти возможности в разделе «Форматирование строк» (см. с. 146).

Из управляющих последовательностей чаще всего используются символы перевода строки (`\n`) и горизонтальной табуляции (`\t`).

Рассмотрим простейшие способы *ввода с клавиатуры*. В классе Console определены методы ввода строки и отдельного символа, но нет методов, которые позволяют непосредственно считывать с клавиатуры числа. Ввод числовых данных выполняется в два этапа:

1. Символы, представляющие собой число, вводятся с клавиатуры в строковую переменную.
2. Выполняется преобразование из строки в переменную соответствующего типа.

Преобразование можно выполнить либо с помощью специального класса Convert, определенного в пространстве имен System, либо с помощью метода Parse, имеющегося в каждом стандартном арифметическом классе. В листинге 3.10 используются оба способа.

Листинг 3.10. Методы ввода

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            Console.WriteLine( "Введите строку" );
            string s = Console.ReadLine();                  // 1
            Console.WriteLine( "s = " + s );

            Console.WriteLine( "Введите символ" );
            char c = (char)Console.Read();                  // 2
            Console.ReadLine();                             // 3
            Console.WriteLine( "c = " + c );
            string buf;        // строка - буфер для ввода чисел
            Console.WriteLine( "Введите целое число" );
            buf = Console.ReadLine();
            int i = Convert.ToInt32( buf );                 // 4
            Console.WriteLine( i );

            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine();
            double x = Convert.ToDouble( buf );             // 5
            Console.WriteLine( x );

            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine();
            double y = double.Parse( buf );                 // 6
            Console.WriteLine( y );

            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine();
            decimal z = decimal.Parse( buf );               // 7
            Console.WriteLine( z );
        }
    }
}
```

К этому примеру необходимо сделать несколько пояснений. Ввод строки выполняется в операторе 1. Длина строки не ограничена, ввод выполняется до символа перевода строки.

Ввод символа выполняется с помощью метода `Read`, который считывает один символ из входного потока (оператор 2). Метод возвращает значение типа `int`, представляющее собой код символа, или –1, если символов во входном потоке нет (например, пользователь нажал клавишу Enter). Поскольку нам требуется не `int`, а `char`, а неявного преобразования от `int` к `char` не существует, приходится применить операцию явного преобразования типа, которая описана в разделе «Явное преобразование типа» (см. с. 49).

За оператором 2 записан оператор 3, который считывает остаток строки и никуда его не передает. Это необходимо потому, что ввод данных выполняется через *буфер* — специальную область оперативной памяти. Фактически, данные сначала заносятся в буфер, а затем считываются оттуда процедурами ввода. Занесение в буфер выполняется по нажатию клавиши Enter вместе с ее кодом. Метод `Read`, в отличие от `ReadLine`, не очищает буфер, поэтому следующий после него ввод будет выполняться с того места, на котором закончился предыдущий.

В операторах 4 и 5 используются методы класса `Convert`, в операторах 6 и 7 — методы `Parse` классов `Double` и `Decimal` библиотеки .NET, которые используются здесь через имена типов C# `double` и `decimal`.

ВНИМАНИЕ

При вводе вещественных чисел дробная часть отделяется от целой с помощью запятой, а не точки. Иными словами, при вводе используются правила операционной системы, а не языка программирования. Допускается задавать числа с порядком, например, 1,95e-8.

Если вводимые с клавиатуры символы нельзя интерпретировать как вещественное число, генерируется исключение[1].

Ввод-вывод в файлы

При отладке даже небольших программ может потребоваться их выполнить не раз, не два и даже не десять. При этом ввод исходных данных может стать утомительным и испортить все удовольствие от процесса[2]. Удобно заранее подготовить исходные данные в текстовом файле и считывать их в программе. Кроме того, это дает возможность не торопясь продумать, какие исходные данные требуется ввести для полной проверки программы, и заранее рассчитать, что должно получиться в результате.

[1] При этом система задает вопрос, хотите ли вы воспользоваться услугами отладчика. От этого предложения следует отказаться, поскольку на данном уровне освоения языка предоставляемые отладчиком сведения будут бесполезны.

[2] А удовольствие — необходимое условие написания хорошей программы!

Вывод из программы тоже бывает полезно выполнить не на экран, а в текстовый файл для последующего неспешного анализа и распечатки. Работа с файлами подробно рассматривается в главе 11, а здесь приводятся лишь образцы для использования в программах. В листинге 3.11 приведена версия программы из листинга 3.9, выполняющая вывод не на экран, а в текстовый файл с именем output.txt. Файл создается в том же каталоге, что и исполняемый файл программы, по умолчанию — ...\ConsoleApplication1\bin\Debug.

Листинг 3.11. Вывод в текстовый файл

```
using System;
using System.IO;                                                    // 1
namespace ConsoleApplication1
{    class Class1
    {    static void Main()
        {
            StreamWriter f = new StreamWriter( "output.txt" );      // 2
            int     i = 3;
            double  y = 4.12;
            decimal d = 600m;
            string  s = "Вася";

            f.WriteLine( "i = " + i );                              // 3
            f.WriteLine( "y = {0} \nd = {1}", y, d );               // 4
            f.WriteLine( "s = " + s );                              // 5

            f.Close();                                              // 6
        }
    }
}
```

Для того чтобы использовать в программе файлы, необходимо:

1. Подключить пространство имен, в котором описываются стандартные классы для работы с файлами (оператор 1).
2. Объявить файловую переменную и связать ее с файлом на диске (оператор 2).
3. Выполнить операции ввода-вывода (операторы 3–5).
4. Закрыть файл (оператор 6).

СОВЕТ

При отладке программы бывает удобно выводить одну и ту же информацию и на экран, и в текстовый файл. Для этого соответствующие операторы дублируют.

Ввод данных из файла выполняется аналогично. В листинге 3.12 приведена программа, аналогичная листингу 3.10, но ввод выполняется из файла с именем input.txt, расположенного в каталоге D:\C#. Естественно, из программы убраны все приглашения к вводу.

Текстовый файл можно создать с помощью любого текстового редактора, но удобнее использовать Visual Studio.NET. Для этого следует выбрать в меню команду File ▸ New ▸ File... и в появившемся диалоговом окне выбрать тип файла Text File.

Листинг 3.12. Ввод из текстового файла

```
using System;
using System.IO;
namespace ConsoleApplication1
{    class Class1
    {    static void Main()
        {
            StreamReader f = new StreamReader( "d:\\C#\\input.txt" );
            string s = f.ReadLine();
            Console.WriteLine( "s = " + s );

            char  c = (char)f.Read();
            f.ReadLine();
            Console.WriteLine( "c = " + c );

            string buf;
            buf = f.ReadLine();
            int i = Convert.ToInt32( buf );
            Console.WriteLine( i );

            buf = f.ReadLine();
            double x = Convert.ToDouble( buf );
            Console.WriteLine( x );

            buf = f.ReadLine();
            double y = double.Parse( buf );
            Console.WriteLine( y );

            buf = f.ReadLine();
            decimal z = decimal.Parse( buf );
            Console.WriteLine( z );
            f.Close();
        }
    }
}
```

Математические функции — класс Math

В выражениях часто используются математические функции, например синус или возведение в степень. Они реализованы в классе Math, определенном в пространстве имен System. С помощью методов этого класса можно вычислить:

- тригонометрические функции: Sin, Cos, Tan;
- обратные тригонометрические функции: ASin, ACos, ATan, ATan2;

- гиперболические функции: `Tanh`, `Sinh`, `Cosh`;
- экспоненту и логарифмические функции: `Exp`, `Log`, `Log10`;
- модуль (абсолютную величину), квадратный корень, знак: `Abs`, `Sqrt`, `Sign`;
- округление: `Ceiling`, `Floor`, `Round`;
- минимум, максимум: `Min`, `Max`;
- степень, остаток: `Pow`, `IEEERemainder`;
- полное произведение двух целых величин: `BigMul`;
- деление и остаток от деления: `DivRem`.

Кроме того, у класса есть два полезных поля: число π и число *e*. Описание методов и полей приведено в табл. 3.8.

Таблица 3.8. Основные поля и статические методы класса Math

Имя	Описание	Результат	Пояснения
`Abs`	Модуль	Перегружен[1]	$\|x\|$ записывается как `Abs(x)`
`Acos`	Арккосинус[2]	`double`	`Acos(double x)`
`Asin`	Арксинус	`double`	`Asin(double x)`
`Atan`	Арктангенс	`double`	`Atan(double x)`
`Atan2`	Арктангенс	`double`	`Atan2(double x, double y)` — угол, тангенс которого есть результат деления `y` на `x`
`BigMul`	Произведение	`long`	`BigMul(int x, int y)`
`Ceiling`	Округление до большего целого	`double`	`Ceiling(double x)`
`Cos`	Косинус	`double`	`Cos(double x)`
`Cosh`	Гиперболический косинус	`double`	`Cosh(double x)`
`DivRem`	Деление и остаток	Перегружен	`DivRem(x, y, rem)`
`E`	База натурального логарифма (число *e*)	`double`	2,71828182845905
`Exp`	Экспонента	`double`	e^x записывается как `Exp(x)`
`Floor`	Округление до меньшего целого	`double`	`Floor(double x)`
`IEEERemainder`	Остаток от деления	`double`	`IEEERemainder(double x, double y)`
`Log`	Натуральный логарифм	`double`	$\log_e x$ записывается как `Log(x)`

продолжение

[1] Это означает, что существует несколько версий метода для различных типов данных.
[2] Угол задается в радианах.

Таблица 3.8 (*продолжение*)

Имя	Описание	Результат	Пояснения
Log10	Десятичный логарифм	double	$\log_{10} x$ записывается как Log10(x)
Max	Максимум из двух чисел	Перегружен	Max(x, y)
Min	Минимум из двух чисел	Перегружен	Min(x, y)
PI	Значение числа π	double	3,14159265358979
Pow	Возведение в степень	double	x^y записывается как Pow(x, y)
Round	Округление	Перегружен	Round(3.1) даст в результате 3, Round (3.8) даст в результате 4
Sign	Знак числа	int	Аргументы перегружены
Sin	Синус	double	Sin(double x)
Sinh	Гиперболический синус	double	Sinh(double x)
Sqrt	Квадратный корень	double	$\sqrt{x}$ записывается как Sqrt(x)
Tan	Тангенс	double	Tan(double x)
Tanh	Гиперболический тангенс	double	Tanh(double x)

В листинге 3.13 приведен пример применения двух методов класса Math. Остальные методы используются аналогично.

Листинг 3.13. Применение методов класса Math

```
using System;
namespace ConsoleApplication1
{    class Class1
    {    static void Main()
        {
            Console.Write( "Введите x: " );
            string buf = Console.ReadLine();
            double x = double.Parse( buf );
            Console.WriteLine( "Значение sin: " + Math.Sin(x) );

            Console.Write( "Введите y: " );
            buf = Console.ReadLine();
            double y = double.Parse( buf );
            Console.WriteLine( "Максимум : " + Math.Max(x, y) );
        }
    }
}
```

В качестве примера рассмотрим программу расчета по заданной формуле

$$y = \sqrt{\pi \cdot x} - e^{0{,}2\sqrt{\alpha}} + 2\operatorname{tg} 2\alpha + 1{,}6 \cdot 10^3 \cdot \log_{10} x^2.$$

Из формулы видно, что исходными данными для программы являются две величины — x и α. Поскольку их тип и точность представления в условии не оговорены, выберем для них тип double. Программа приведена в листинге 3.14.

Листинг 3.14. Программа расчета по заданной формуле

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {

            Console.WriteLine( "Введите x" );
            double x = Convert.ToDouble( Console.ReadLine() );

            Console.WriteLine( "Введите alfa" );
            double a = Convert.ToDouble(Console.ReadLine() );

            double y = Math.Sqrt( Math.PI * x ) -
                       Math.Exp( 0.2 * Math.Sqrt(a) ) +
                       2 * Math.Tan( 2 * a ) +
                       1.6e3 * Math.Log10( Math.Pow(x, 2) );

            Console.WriteLine( "Для x = {0} и alfa = {1}", x, a );
            Console.WriteLine( "Результат = " + y );
        }
    }
}
```

Итак, к настоящему моменту у вас накопилось достаточно сведений, чтобы писать на C# простейшие линейные программы, выполняющие вычисления по формулам. В следующей главе мы займемся изучением операторов, позволяющих реализовывать более сложные алгоритмы.

Рекомендации по программированию

Приступая к написанию программы, четко определите, что является ее исходными данными и что требуется получить в результате. *Выбирайте тип переменных* с учетом диапазона и требуемой точности представления данных.

Давайте переменным имена, отражающие их назначение. Правильно выбранные имена могут сделать программу в некоторой степени самодокументированной. Неудачные имена, наоборот, служат источником проблем. В именах следует избегать сокращений. Они делают программу менее понятной, к тому же часто легко забыть, как именно было сокращено то или иное слово.

Общая тенденция такая: чем больше область действия переменной, тем более длинное у нее имя. Перед таким именем можно поставить префикс типа (одну или несколько букв, по которым можно определить тип переменной). Напротив, для переменных, вся «жизнь» которых проходит на протяжении нескольких строк кода, лучше обойтись однобуквенными именами типа `i` или `k`.

Имена переменных логического типа, используемые в качестве флагов, должны быть такими, чтобы по ним можно было судить о том, что означают значения `true` и `false`. Например, признак «пусто» лучше описать не как `bool flag`, а как `bool empty`.

ПРИМЕЧАНИЕ

В C# принято называть классы, методы и константы в соответствии с нотацией Паскаля, а локальные переменные — в соответствии с нотацией Camel (см. раздел «Идентификаторы», с. 24).

Переменные желательно инициализировать при их объявлении, а объявлять как можно ближе к месту их непосредственного использования. С другой стороны, удобно все объявления локальных переменных метода располагать в начале блока так, чтобы их было просто найти. При небольших размерах методов оба эти пожелания довольно легко совместить.

Избегайте использования в программе чисел в явном виде. Константы должны иметь осмысленные имена, заданные с помощью ключевого слова `const`. Символическое имя делает программу более понятной, а кроме того, при необходимости изменить значение константы потребуется изменить программу только в одном месте. Конечно, этот совет не относится к константам 0 и 1.

Ввод с клавиатуры предваряйте приглашением, а выводимые значения — пояснениями. Для контроля сразу же после ввода выводите исходные данные на дисплей (по крайней мере, в процессе отладки).

До запуска программы *подготовьте тестовые примеры*, содержащие исходные данные и ожидаемые результаты. Отдельно проверьте реакцию программы на неверные исходные данные.

При записи выражений *обращайте внимание на приоритет операций.* Если в одном выражении соседствует несколько операций одинакового приоритета, операции присваивания и условная операция выполняются справа налево, остальные — слева направо. Для изменения порядка выполнения операций используйте круглые скобки.

Тщательно форматируйте текст программы так, чтобы его было удобно читать. Ставьте пробелы после знаков препинания, отделяйте пробелами знаки операций, не пишите много операторов в одной строке, используйте комментарии и пустые строки для разделения логически законченных фрагментов программы.

Глава 4
Операторы

Операторы языка вместе с его типами данных определяют круг задач, которые можно решать с помощью этого языка. C# реализует типичный набор операторов для языка программирования общего назначения. В этой главе рассматриваются основные операторы C#, составляющие так называемые базовые конструкции структурного программирования.

Структурное программирование — это технология создания программ, позволяющая путем соблюдения определенных правил сократить время разработки и уменьшить количество ошибок, а также облегчить возможность модификации программы.

Структурный подход, сформировавшийся в 60–70-х годах прошлого столетия, позволил довольно успешно создавать достаточно крупные проекты, но сложность программного обеспечения продолжала возрастать, и требовались все более развитые средства ее преодоления. Идеи структурного программирования получили свое дальнейшее развитие в *объектно-ориентированном программировании* — технологии, позволяющей достичь простоты структуры и управляемости очень больших программных систем.

Несмотря на то что язык C# реализует объектно-ориентированную парадигму, принципы структурного программирования лежат в основе кодирования каждого метода, каждого фрагмента алгоритма.

Не существует единственного самого лучшего способа создания программ. Для решения задач разного рода и уровня сложности требуется применять разные технологии программирования. В простейших случаях достаточно освоить азы структурного написания программ. Для создания же сложных проектов требуется не только свободно владеть языком в полном объеме, но и иметь представление о принципах проектирования и отладки программ, возможностях библиотеки и т. д. Как правило, чем сложнее задача, тем больше времени требуется на освоение инструментов, необходимых для ее решения.

Выражения, блоки и пустые операторы

Любое *выражение*, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении выражения. Частным случаем выражения является *пустой оператор* ; (он используется, когда по синтаксису оператор требуется, а по смыслу — нет). Примеры:

```
i++;              // выполняется операция инкремента
a *= b + c;       // выполняется умножение с присваиванием
fun( i, k );      // выполняется вызов функции
while( true );    // цикл из пустого оператора (бесконечный)
```

Блок, или *составной оператор*, — это последовательность описаний и операторов, заключенная в фигурные скобки. Блок воспринимается компилятором как один оператор и может использоваться всюду, где синтаксис требует одного оператора, а алгоритм — нескольких. Блок может содержать один оператор или быть пустым.

Операторы ветвления

Операторы ветвления `if` и `switch` применяются для того чтобы в зависимости от конкретных значений исходных данных обеспечить выполнение разных последовательностей операторов. Оператор `if` обеспечивает передачу управления на одну из двух ветвей вычислений, а оператор `switch` — на одну из произвольного числа ветвей.

Условный оператор if

Условный оператор `if` используется для разветвления процесса вычислений на два направления. Структурная схема оператора приведена на рис. 4.1.

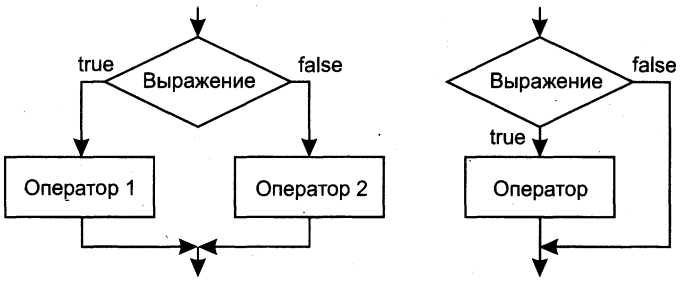


Рис. 4.1. Структурная схема условного оператора

Формат оператора:

```
if ( логическое_выражение ) оператор_1; [ else оператор_2; ]
```

Сначала вычисляется логическое выражение. Если оно имеет значение true, выполняется первый оператор, иначе — второй. После этого управление передается на оператор, следующий за условным. Ветвь else может отсутствовать.

ПРИМЕЧАНИЕ

Операторы, входящие в условный, не должны иметь метку и не могут быть описаниями.

Если в какой-либо ветви требуется выполнить несколько операторов, их необходимо заключить в *блок*, иначе компилятор не сможет понять, где заканчивается ветвление. Блок может содержать любые операторы, в том числе описания и другие условные операторы (но не может состоять из одних описаний). Необходимо учитывать, что переменная, описанная в блоке, вне блока не существует.

Примеры условных операторов:

```
if ( a < 0 ) b = 1;                                              // 1
if ( a < b && ( a > d || a == 0 ) ) b++; else { b *= a; a = 0; }  // 2
if ( a < b ) if ( a < c ) m = a; else m = c;
else         if ( b < c ) m = b; else m = c;                     // 3
if ( b > a ) max = b; else max = a;                              // 4
```

В примере 1 отсутствует ветвь else. Подобная конструкция реализует пропуск оператора, поскольку присваивание либо выполняется, либо пропускается в зависимости от выполнения условия.

Если требуется проверить несколько условий, их объединяют знаками логических условных операций. Например, выражение *в примере 2* будет истинно в том случае, если выполнится одновременно условие a < b и одно из условий в скобках. Если опустить внутренние скобки, будет выполнено сначала логическое И, а потом — ИЛИ.

Оператор *в примере 3* вычисляет наибольшее значение из трех переменных. Обратите внимание на то, что компилятор относит часть else к ближайшему ключевому слову if.

Конструкции, подобные оператору *в примере 4* (вычисляется наибольшее значение из двух переменных), проще и нагляднее записывать в виде условной операции, в данном случае следующей:

```
max = b > a ? b : a;
```

ПРИМЕЧАНИЕ

Распространенная ошибка начинающих — неверная запись проверки на принадлежность диапазону. Например, чтобы проверить условие 0 < x < 1, нельзя записать его в условном операторе непосредственно, так как каждая операция отношения должна иметь два операнда. Правильный способ записи: if(0 < x && x < 1)...

В качестве примера подсчитаем количество очков после выстрела по мишени, изображенной на рис. 4.2.

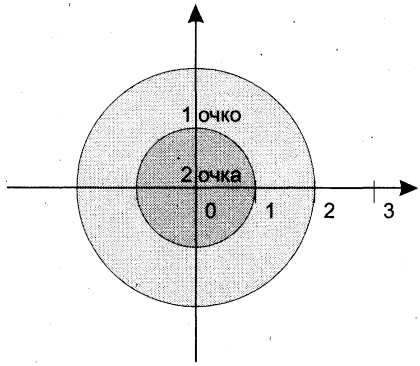


Рис. 4.2. Мишень

Как уже говорилось, тип переменных выбирается, исходя из их назначения. Координаты выстрела нельзя представить целыми величинами, так как это приведет к потере точности результата, а счетчик очков не имеет смысла описывать как вещественный. Программа приведена в листинге 4.1.

Листинг 4.1. Выстрел по мишени

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            Console.WriteLine( "Введите координату x" );
            double x = Convert.ToDouble( Console.ReadLine() );

            Console.WriteLine( "Введите координату y" );
            double y = double.Parse( Console.ReadLine() );

            int kol = 0;
            if      ( x * x + y * y < 1 ) kol = 2;
            else if ( x * x + y * y < 4 ) kol = 1;
            Console.WriteLine( "Результат = {0} очков", kol );
        }
    }
}
```

Даже такую простую программу можно еще упростить с помощью промежуточной переменной и записи условия в виде двух последовательных, а не вложенных, операторов if:

```
double temp = x * x + y * y;
int kol = 0;
if ( temp < 4 ) kol = 1;
```

```
if ( temp < 1 ) kol = 2;
Console.WriteLine( "Результат = {0} очков", kol );
```

Обратите внимание на то, что в первом варианте программы второй оператор `if` выполняется только в случае невыполнения условия в первом условном операторе, а во втором варианте оба условных оператора выполняются последовательно, один за другим.

Следует избегать проверки вещественных величин на равенство, вместо этого лучше сравнивать модуль их разности с некоторым малым числом. Это связано с погрешностью представления вещественных значений в памяти:

```
float a, b; …
if  ( a == b ) …                    // не рекомендуется!
if  ( Math.Abs(a - b) < 1e-6 ) …    // надежно!
```

Значение величины, с которой сравнивается модуль разности, следует выбирать в зависимости от решаемой задачи и точности участвующих в выражении переменных. Снизу эта величина ограничена определенной в классах `Single` и `Double` константой `Epsilon` (это минимально возможное значение переменной такое, что `1.0 + Epsilon != 1.0`).

Оператор выбора switch

Оператор `switch` (переключатель) предназначен для разветвления процесса вычислений на несколько направлений. Структурная схема оператора приведена на рис. 4.3.

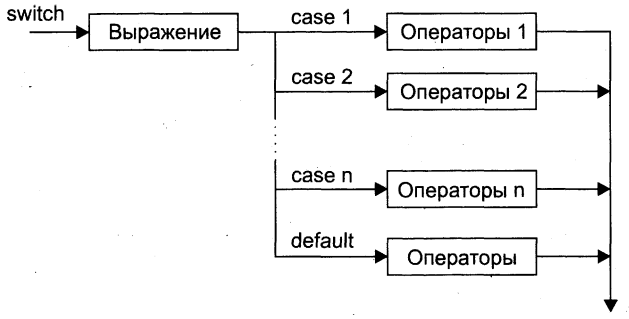


Рис. 4.3. Структурная схема оператора switch

Формат оператора:

```
switch ( выражение ){
    case константное_выражение_1: [ список_операторов_1 ]
    case константное_выражение_2: [ список_операторов_2 ]
    ...
    case константное_выражение_n: [ список_операторов_n ]
    [ default: операторы ]
}
```

Выполнение оператора начинается с вычисления выражения. Тип выражения чаще всего целочисленный (включая char) или строковый[1]. Затем управление передается первому оператору из списка, помеченному константным выражением, значение которого совпало с вычисленным.

Все константные выражения должны быть неявно приводимы к типу выражения в скобках. Если совпадения не произошло, выполняются операторы, расположенные после слова default (а при его отсутствии управление передается следующему за switch оператору).

Каждая ветвь переключателя должна заканчиваться явным *оператором перехода*, а именно оператором break, goto или return:

- оператор break выполняет выход из самого внутреннего из объемлющих его операторов switch, for, while и do (см. раздел «Оператор break», с. 84);
- оператор goto выполняет переход на указанную после него метку, обычно это метка case одной из нижележащих ветвей оператора switch (см. раздел «Оператор goto», с. 83);
- оператор return выполняет выход из функции, в теле которой он записан (см. раздел «Оператор return», с. 87).

Оператор goto обычно используют для последовательного выполнения нескольких ветвей переключателя, однако поскольку это нарушает читабельность программы, такого решения следует избегать.

В листинге 4.2 приведен пример программы, реализующей простейший калькулятор на четыре действия.

Листинг 4.2. Калькулятор на четыре действия

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double a, b, res;

            Console.WriteLine( "Введите первый операнд:" );
            a = double.Parse( Console.ReadLine() );

            Console.WriteLine( "Введите знак операции" );
            char op = (char)Console.Read();
            Console.ReadLine();

            Console.WriteLine( "Введите второй операнд:" );
            b = double.Parse( Console.ReadLine() );
```

[1] В общем случае выражение может быть любого типа, для которого существует неявное преобразование к указанным, а также перечисляемого типа.

```
            bool ok = true;
            switch (op)
            {
                case '+' : res = a + b; break;
                case '-' : res = a - b; break;
                case '*' : res = a * b; break;
                case '/' : res = a / b; break;
                default  : res = double.NaN; ok = false; break;
            }
            if (ok) Console.WriteLine( "Результат: " + res );
            else    Console.WriteLine( "Недопустимая операция" );
        }
    }
}
```

СОВЕТ

Хотя наличие ветви `default` и не обязательно, рекомендуется всегда обрабатывать случай, когда значение выражения не совпадает ни с одной из констант. Это облегчает поиск ошибок при отладке программы.

Оператор `switch` предпочтительнее оператора `if` в тех случаях, когда в программе требуется разветвить вычисления на количество направлений большее двух и выражение, по значению которого производится переход на ту или иную ветвь, не является вещественным. Часто это справедливо даже для двух ветвей, поскольку повышает наглядность программы.

Операторы цикла

Операторы цикла используются для вычислений, повторяющихся многократно. В C# имеется четыре вида циклов: цикл с предусловием `while`, цикл с постусловием `repeat`, цикл с параметром `for` и цикл перебора `foreach`. Каждый из них состоит из определенной последовательности операторов.

Блок, ради выполнения которого и организуется цикл, называется *телом цикла*. Остальные операторы служат для управления процессом повторения вычислений: это начальные установки, проверка условия продолжения цикла и модификация параметра цикла (рис. 4.4). Один проход цикла называется *итерацией*.

Начальные установки служат для того, чтобы до входа в цикл задать значения переменных, которые в нем используются.

Проверка условия продолжения цикла выполняется на каждой итерации либо до тела цикла (тогда говорят о цикле *с предусловием*, схема которого показана на рис. 4.4, *а*), либо после тела цикла (цикл *с постусловием*, рис. 4.4, *б*). Разница между ними состоит в том, что тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Проверка необходимости выполнения цикла с предусловием делается до тела цикла, поэтому возможно, что он не выполнится ни разу.

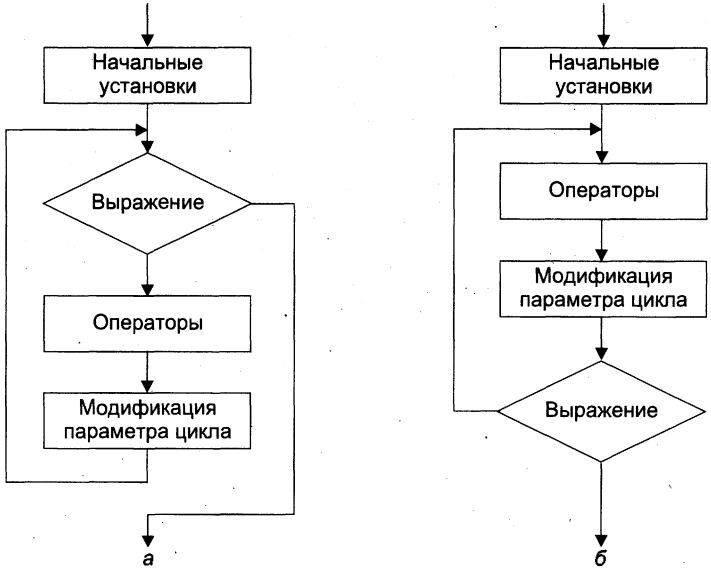


Рис. 4.4. Структурные схемы операторов цикла

Параметром цикла называется переменная, которая используется при проверке условия продолжения цикла и принудительно изменяется на каждой итерации, причем, как правило, на одну и ту же величину. Если параметр цикла целочисленный, он называется *счетчиком цикла*. Количество повторений такого цикла можно определить заранее. Параметр есть не у всякого цикла. В так называемом *итеративном* цикле условие продолжения содержит переменные, значения которых изменяются в цикле по рекуррентным формулам[1].

Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом. Для этого служат операторы `break`, `continue`, `return` и `goto` (см. раздел «Операторы передачи управления», с. 83). Передавать управление извне внутрь цикла запрещается (при этом возникает ошибка компиляции).

Цикл с предусловием while

Формат оператора прост:

```
while ( выражение ) оператор
```

Выражение должно быть логического типа. Например, это может быть операция отношения или просто логическая переменная. Если результат вычисления выражения равен `true`, выполняется простой или составной оператор (блок). Эти действия повторяются до того момента, пока результатом выражения не станет значение `false`. После окончания цикла управление передается на следующий за ним оператор.

[1] Рекуррентной называется формула, в которой новое значение переменной вычисляется с использованием ее предыдущего значения.

Выражение вычисляется перед каждой итерацией цикла. Если при первой проверке выражение равно false, цикл не выполнится ни разу.

ВНИМАНИЕ

Если в теле цикла необходимо выполнить более одного оператора, необходимо заключить их в блок с помощью фигурных скобок.

В качестве примера рассмотрим программу, выводящую для аргумента x, изменяющегося в заданных пределах с заданным шагом, таблицу значений следующей функции:

$$y = \begin{cases} t, & x < 0 \\ tx, & 0 \le x < 10 \\ 2t, & x \ge 10 \end{cases}.$$

Назовем начальное значение аргумента Xn, конечное значение аргумента — Xk, шаг изменения аргумента — dX и параметр t. Все величины вещественные. Программа должна выводить таблицу, состоящую из двух столбцов: значений аргумента и соответствующих им значений функции.

Опишем алгоритм в словесной форме:

1. Взять первое значение аргумента.
2. Определить, какому из интервалов оно принадлежит, и вычислить значение функции по соответствующей формуле.
3. Вывести строку таблицы.
4. Перейти к следующему значению аргумента.
5. Если оно не превышает конечное значение, повторить шаги 2–4, иначе закончить.

Шаги 2–4 повторяются многократно, поэтому для их выполнения надо организовать цикл. Текст программы приведен в листинге 4.3. Строки программы помечены соответствующими номерами шагов алгоритма. Обратите внимание на то, что условие продолжения цикла записано в его заголовке и проверяется до входа в цикл. Таким образом, если задать конечное значение аргумента, меньшее начального, даже при отрицательном шаге цикл не будет выполнен ни разу.

Листинг 4.3. Таблица значений функции, полученных с использованием цикла while

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double Xn = -2, Xk = 12, dX = 2, t = 2, y;
            Console.WriteLine( "|   x    |    y    |" ); // заголовок таблицы

            double x = Xn;                                          // 1
            while ( x <= Xk )                                       // 5
            {
                y = t;                                              // 2
```

продолжение

Листинг 4.3 (*продолжение*)

```
                if ( x >= 0 && x < 10 ) y = t * x;                    // 2
                if ( x >= 10 )          y = 2 * t;                    // 2
                Console.WriteLine( "| {0,6} | {1,6} |", x, y );      // 3
                x += dX;                                              // 4
            }
        }
    }
}
```

ПРИМЕЧАНИЕ

При выводе использованы дополнительные поля спецификаций формата, определяющие желаемую ширину поля вывода под переменную (в данном случае — 6 позиций). В результате колонки при выводе таблицы получаются ровными. Описание спецификаций формата приведено в приложении.

Параметром этого цикла, то есть переменной, управляющей его выполнением, является x. Блок модификации параметра цикла представлен оператором, выполняющимся на шаге 4. Для перехода к следующему значению аргумента текущее значение наращивается на величину шага и заносится в ту же переменную. Начинающие часто забывают про модификацию параметра, в результате программа «зацикливается». Если с вами произошла такая неприятность, попробуйте для завершения программы нажать клавиши Ctrl+Break, а впредь перед запуском программы проверяйте:

- присвоено ли параметру цикла верное начальное значение;
- изменяется ли параметр цикла на каждой итерации цикла;
- верно ли записано условие продолжения цикла.

Распространенным приемом программирования является организация бесконечного цикла с заголовком `while (true)` и принудительным выходом из тела цикла по выполнению какого-либо условия с помощью операторов передачи управления. В листинге 4.4 приведен пример использования бесконечного цикла для организации меню программы.

Листинг 4.4. Организация меню

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            string buf;
            while ( true )
            {
                Console.WriteLine( "1 - пункт_1, 2 - пункт_2, 3 - выход" );
                buf = Console.ReadLine();
                switch ( buf )
                {
                    case "1" :          // Вставить код обработки пункта 1
                          Console.WriteLine( "отладка - пункт_1" );
```

```
                    break;

                case "2" :          // Вставить код обработки пункта 2
                    Console.WriteLine( "отладка - пункт_2" );
                    break;

                case "3" : return;

                default  : Console.WriteLine( "не попал по клавише!" );
                    break;
            }
        }
    }
}
```

Цикл с постусловием do

Цикл с постусловием реализует структурную схему, приведенную на рис. 4.4, *б*, и имеет вид

```
do оператор while выражение;
```

Сначала выполняется простой или составной оператор, образующий тело цикла, а затем вычисляется выражение (оно должно иметь тип `bool`). Если выражение истинно, тело цикла выполняется еще раз и проверка повторяется. Цикл завершается, когда выражение станет равным `false` или в теле цикла будет выполнен какой-либо оператор передачи управления.

Этот вид цикла применяется в тех случаях, когда тело цикла необходимо обязательно выполнить хотя бы один раз, например, если в цикле вводятся данные и выполняется их проверка. Если же такой необходимости нет, предпочтительнее пользоваться циклом с предусловием.

Пример программы, выполняющей проверку ввода, приведен в листинге 4.5.

Листинг 4.5. Проверка ввода

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            char answer;
            do
            {
                Console.WriteLine( "Купи слоника, а?" );
                answer = (char) Console.Read();
                Console.ReadLine();
            } while ( answer != 'y' );
        }
    }
}
```

Рассмотрим еще один пример применения цикла с постусловием — программу, определяющую корень уравнения $\cos(x) = x$ методом деления пополам с точностью 0,0001.

Исходные данные для этой задачи — точность, результат — число, представляющее собой корень уравнения[1]. Оба значения имеют вещественный тип.

Суть метода деления пополам очень проста. Задается интервал, в котором есть ровно один корень (следовательно, на концах этого интервала функция имеет значения разных знаков). Вычисляется значение функции в середине этого интервала. Если оно того же знака, что и значение на левом конце интервала, значит, корень находится в правой половине интервала, иначе — в левой. Процесс повторяется для найденной половины интервала до тех пор, пока его длина не станет меньше заданной точности.

В приведенной далее программе (листинг 4.6) исходный интервал задан с помощью констант, значения которых взяты из графика функции. Для уравнений, имеющих несколько корней, можно написать дополнительную программу, определяющую (путем вычисления и анализа таблицы значений функции) интервалы, содержащие ровно один корень[2].

Листинг 4.6. Вычисление корня нелинейного уравнения

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double x, left = 0, right = 1;
            do
            {
                x = ( left + right ) / 2;
                if ( ( Math.Cos(x) - x ) * ( Math.Cos(left) - left ) < 0 )
                    right = x;
                else left = x;
            } while ( Math.Abs( right - left ) < 1e-4 );
            Console.WriteLine( "Корень равен " + x );
        }
    }
}
```

В эту программу для надежности очень полезно добавить подсчет количества выполненных итераций и принудительный выход из цикла при превышении их разумного количества.

---

[1] Корнем уравнения называется значение, при подстановке которого в уравнение оно превращается в тождество.

[2] Программа печати таблицы значений функции приведена в листинге 4.3.

Цикл с параметром for

Цикл с параметром имеет следующий формат:

```
for ( инициализация; выражение; модификации ) оператор;
```

Инициализация служит для объявления величин, используемых в цикле, и присвоения им начальных значений. В этой части можно записать несколько операторов, разделенных запятой, например:

```
for ( int i = 0, j = 20; ...
int k, m;
for ( k = 1, m = 0; ...
```

Областью действия переменных, объявленных в части инициализации цикла, является цикл. Инициализация выполняется один раз в начале исполнения цикла.

Выражение типа bool определяет условие выполнения цикла: если его результат равен true, цикл выполняется. Цикл с параметром реализован как цикл с предусловием.

Модификации выполняются после каждой итерации цикла и служат обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую, например:

```
for ( int i = 0, j = 20; i < 5 && j > 10; i++, j-- ) ...
```

Простой или составной *оператор* представляет собой тело цикла. Любая из частей оператора for может быть опущена (но точки с запятой надо оставить на своих местах!).

Для примера вычислим сумму чисел от 1 до 100:

```
int s = 0;
for ( int i = 1; i <= 100; i++ ) s += i;
```

В листинге 4.7 приведена программа, выводящая таблицу значений функции из листинга 4.3.

Листинг 4.7. Таблица значений функции, полученных с использованием цикла for

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double Xn = -2, Xk = 12, dX = 2, t = 2, y;
            Console.WriteLine( "|   x    |    y   |";   // заголовок таблицы

            for ( double x = Xn; x <= Xk; x += dX )          // 1, 4, 5
            {
                y = t;                                       // 2
                if ( x >= 0 && x < 10 ) y = t * x;           // 2
```

продолжение

Листинг 4.7 (*продолжение*)

```
                if ( x >= 10 )          y = 2 * t;                  // 2
                Console:WriteLine( "| {0,6} | {1,6} |", x, y );    // 3
            }
        }
    }
}
```

Как видите, в этом варианте программы все управление циклом сосредоточено в его заголовке. Это делает программу понятней. Кроме того, областью действия служебной переменной `x` является цикл, а не вся функция, как это было в листинге 4.3, что предпочтительнее, поскольку переменная `x` вне цикла не требуется.

СОВЕТ

В общем случае надо стремиться к минимизации области действия переменных. Это облегчает поиск ошибок в программе.

Любой цикл `while` может быть приведен к эквивалентному ему циклу `for` и наоборот. Например, два следующих цикла эквивалентны:

Цикл `for`:

```
for ( b1; b2; b3 ) оператор;
```

Цикл `while`:

```
b1;
while ( b2 )
{
    оператор;
    b3
}
```

Цикл перебора foreach

Цикл `foreach` используется для просмотра всех объектов из некоторой группы данных, например массива, списка или другого контейнера. Он будет рассмотрен, когда у нас появится в нем необходимость, а именно в разделе «Оператор foreach» (см. с. 136).

Рекомендации по выбору оператора цикла

Операторы цикла взаимозаменяемы, но можно привести некоторые *рекомендации* по выбору наилучшего в каждом конкретном случае.

Оператор `do while` обычно используют, когда цикл требуется обязательно выполнить хотя бы раз, например, если в цикле производится ввод данных.

Оператором `while` удобнее пользоваться в тех случаях, когда либо число итераций заранее неизвестно, либо очевидных параметров цикла нет, либо модификацию параметров удобнее записывать не в конце тела цикла.

Оператор foreach применяют для просмотра элементов различных коллекций объектов.

Оператор for предпочтительнее в большинстве остальных случаев. Однозначно — для организации циклов со счетчиками, то есть с целочисленными переменными, которые изменяют свое значение при каждом проходе цикла регулярным образом (например, увеличиваются на 1).

Начинающие часто делают ошибки при записи циклов. Чтобы избежать этих ошибок, рекомендуется:

- проверить, всем ли переменным, встречающимся в правой части операторов присваивания в теле цикла, присвоены до этого правильные начальные значения (а также возможно ли выполнение других операторов);
- проверить, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла;
- предусмотреть аварийный выход из цикла по достижении некоторого количества итераций (пример приведен в следующем разделе);
- и, конечно, не забывать о том, что если в теле цикла требуется выполнить более одного оператора, нужно заключать их в фигурные скобки.

Операторы передачи управления

В C# есть пять операторов, изменяющих естественный порядок выполнения вычислений:

- оператор безусловного перехода goto;
- оператор выхода из цикла break;
- оператор перехода к следующей итерации цикла continue;
- оператор возврата из функции return;
- оператор генерации исключения throw.

Эти операторы могут передать управление в пределах блока, в котором они использованы, и за его пределы. Передавать управление внутрь другого блока запрещается.

Первые четыре оператора рассматриваются в этом разделе, а оператор throw — далее в этой главе на с. 93.

Оператор goto

Оператор безусловного перехода goto используется в одной из трех форм:

```
goto метка;
goto case константное_выражение;
goto default;
```

В теле той же функции должна присутствовать ровно одна конструкция вида

```
метка: оператор;
```

Оператор `goto метка` передает управление на помеченный оператор. *Метка* — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан. Метка должна находиться в той же области видимости, что и оператор перехода. Использование этой формы оператора безусловного перехода оправдано в двух случаях:

- принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей;
- переход из нескольких точек функции вниз по тексту в одну точку (например, если перед выходом из функции необходимо всегда выполнять какие-либо действия).

В остальных случаях для записи любого алгоритма существуют более подходящие средства, а использование оператора `goto` приводит только к усложнению структуры программы и затруднению отладки. Применение этого оператора нарушает принципы структурного и модульного программирования, по которым все блоки, образующие программу, должны иметь только один вход и один выход.

Вторая и третья формы оператора `goto` используются в теле оператора выбора `switch`. Оператор `goto case константное_выражение` передает управление на соответствующую константному выражению ветвь, а оператор `goto default` — на ветвь `default`. Надо отметить, что реализация оператора выбора в C# на редкость неудачна, и наличие в нем оператора безусловного перехода затрудняет понимание программы, поэтому лучше обходиться без него.

Оператор break

Оператор `break` используется внутри операторов цикла или выбора для перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится оператор `break`.

Для примера рассмотрим программу вычисления значения функциифункции sin x (синус) с точностью $\varepsilon = 10^{-6}$ с помощью бесконечного ряда Тейлора по формуле

$$y = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots + (-1)^n \frac{x^{2n-1}}{(2n-1)!} + \ldots$$

Этот ряд сходится при $|x| < \infty$. Точность достигается при $|Rn| < \varepsilon$, где *Rn* —остаточный член ряда, который для данного ряда можно заменить величиной *Cn* очередного члена ряда, прибавляемого к сумме.

Алгоритм решения задачи выглядит так: задать начальное значение суммы ряда, а затем многократно вычислять очередной член ряда и добавлять его к ранее найденной сумме. Вычисления заканчиваются, когда абсолютная величина очередного члена ряда станет меньше заданной точности.

До выполнения программы предсказать, сколько членов ряда потребуется просуммировать, невозможно. В цикле такого рода есть опасность, что он никогда не завершится — как из-за возможных ошибок в вычислениях, так и из-за ограниченной

области сходимости ряда (данный ряд сходится на всей числовой оси, но существуют ряды Тейлора, которые сходятся только для определенного интервала значений аргумента). Поэтому для надежности программы необходимо предусмотреть аварийный выход из цикла с печатью предупреждающего сообщения по достижении некоторого максимально допустимого количества итераций. Для выхода из цикла применяется оператор break.

Прямое вычисление члена ряда по приведенной общей формуле, когда x возводится в степень, вычисляется факториал, а затем числитель делится на знаменатель, имеет два недостатка, которые делают этот способ непригодным. Первый недостаток — большая погрешность вычислений. При возведении в степень и вычислении факториала можно получить очень большие числа, при делении которых друг на друга произойдет потеря точности, поскольку количество значащих цифр, хранимых в ячейке памяти, ограничено[1]. Второй недостаток связан с эффективностью вычислений: как легко заметить, при вычислении очередного члена ряда нам уже известен предыдущий, поэтому вычислять каждый член ряда «от печки» нерационально.

Для уменьшения количества выполняемых действий следует воспользоваться рекуррентной формулой получения последующего члена ряда через предыдущий:

$$C_{n+1} = C_n \cdot T,$$

где T — некоторый множитель. Подставив в эту формулу C_n и C_{n+1}, получим выражение для вычисления T:

$$T = \frac{C_{n+1}}{C_n} = \frac{(-1)^{n+1} x^{2(n+1)-1} (2n-1)!}{(-1)^n x^{2n-1} (2(n+1)-1)!} = \frac{x^2}{2n(2n+1)}.$$

В листинге 4.8 приведен текст программы с комментариями.

Листинг 4.8. Вычисление суммы бесконечного ряда

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double e = 1e-6;
            const int MaxIter = 500;      // ограничитель количества итераций
            Console.WriteLine( "Введите аргумент:" );
            double x = Convert.ToDouble( Console.ReadLine() );

            bool done = true;                  // признак достижения точности
            double ch = x, y = ch;
            for ( int n = 1; Math.Abs(ch) > e; n++ )
            {
```

продолжение

[1] Кроме того, большие числа могут переполнить разрядную сетку.

Листинг 4.8 (*продолжение*)

```
                ch *= - x * x /(2 * n * (2 * n + 1)); // очередной член ряда
                y += ch;                    // добавление члена ряда к сумме
                if ( n > MaxIter ) { done = false; break; }
            }
            if ( done ) Console.WriteLine( "Сумма ряда - " + y );
            else        Console.WriteLine( "Ряд расходится" );
        }
    }
}
```

Получение суммы бесконечного ряда — пример вычислений, которые принципиально невозможно выполнить точно. В данном случае мы задавали желаемую погрешность вычислений с помощью значения ε. Это значение не может быть меньше, чем самое малое число, представимое с помощью переменной типа `double`, но при задании такого значения точность результата фактически будет гораздо ниже из-за погрешностей, возникающих при вычислениях. Они связаны с конечностью разрядной сетки.

В общем случае *погрешность результата* складывается из нескольких частей:

- погрешность постановки задачи (возникает при упрощении задачи);
- начальная погрешность (точность представления исходных данных);
- погрешность метода (при использовании приближенных методов решения задачи);
- погрешности округления и вычисления (поскольку величины хранятся в ограниченном количестве разрядов).

Специфика машинных вычислений состоит в том, что алгоритм, безупречный с точки зрения математики, при реализации без учета возможных погрешностей может привести к получению результатов, не содержащих ни одной верной значащей цифры! Это происходит, например, при вычитании двух близких значений или при работе с очень большими или очень малыми числами.

Оператор continue

Оператор перехода к следующей итерации текущего цикла `continue` пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации.

Перепишем основной цикл листинга 4.8 с применением оператора `continue`:

```
for ( int n = 0; Math.Abs(ch) > e; n++ )
{
    ch *= x * x / ( 2 * n + 1 ) / ( 2 * n + 2 );
    y += ch;
    if ( n <= MaxIter ) continue;
    done = false; break;
}
```

Оператор return

Оператор возврата из функции return завершает выполнение функции и передает управление в точку ее вызова. Синтаксис оператора:

```
return [ выражение ];
```

Тип выражения должен иметь неявное преобразование к типу функции. Если тип возвращаемого функцией значения описан как void, выражение должно отсутствовать.

Базовые конструкции структурного программирования

Главное требование, которому должна удовлетворять программа, — работать в полном соответствии со спецификацией и адекватно реагировать на любые действия пользователя. Кроме этого, программа должна быть выпущена точно к заявленному сроку и допускать оперативное внесение необходимых изменений и дополнений.

Иными словами, современные критерии качества программы — это, прежде всего, *надежность*, а также возможность точно *планировать* производство программы и ее *сопровождение*. Для достижения этих целей программа должна иметь простую структуру, быть читабельной и легко модифицируемой. Технология структурного программирования позволяет создавать как раз такие программы небольшого и среднего объема. Для разработки более сложных комплексов требуется применять объектно-ориентированное программирование.

В C# идеи структурного программирования используются на самом низком уровне — при написании методов объектов. Доказано, что любой алгоритм можно реализовать только из трех структур, называемых *базовыми конструкциями* структурного программирования, — это следование, ветвление и цикл.

Следованием называется конструкция, реализующая последовательное выполнение двух или более операторов (простых или составных). *Ветвление* задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия. *Цикл* реализует многократное выполнение оператора. Базовые конструкции приведены на рис. 4.5.

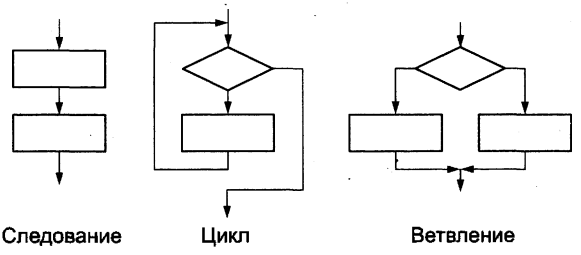


Рис. 4.5. Базовые конструкции структурного программирования

Особенностью базовых конструкций является то, что любая из них имеет только один вход и один выход, поэтому конструкции могут вкладываться друг в друга произвольным образом, например, цикл может содержать следование из двух ветвлений, каждое из которых включает вложенные циклы (рис. 4.6).

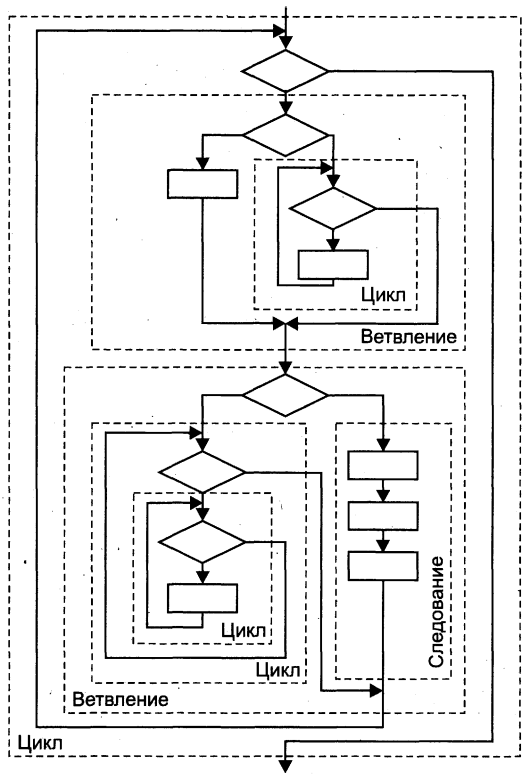


Рис. 4.6. Вложение базовых конструкций

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать (а программы чаще приходится читать, чем писать), отлаживать и при необходимости вносить в нее изменения.

В большинстве языков высокого уровня существует несколько реализаций базовых конструкций; в C# есть четыре вида циклов и два вида ветвлений (на два и на произвольное количество направлений). Они введены для удобства программирования, и в каждом случае надо выбирать наиболее подходящие средства. Главное, о чем нужно помнить даже при написании самых простых программ, — они должны состоять из четкой последовательности блоков строго определенной структуры. «Кто ясно мыслит, тот ясно излагает» — практика давно показала, что программы в стиле «поток сознания» нежизнеспособны, не говоря о том, что они просто некрасивы.

Обработка исключительных ситуаций

В языке C# есть операторы, позволяющие обнаруживать и обрабатывать ошибки (исключительные ситуации), возникающие в процессе выполнения программы. Об этом уже упоминалось в разделе «Введение в исключения» (см. с. 46), а сейчас мы рассмотрим механизм обработки исключений более подробно.

Исключительная ситуация, или *исключение*, — это возникновение аварийного события, которое может порождаться некорректным использованием аппаратуры или неправильной работой программы, например делением на ноль или переполнением. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. C# дает программисту возможность восстановить работоспособность программы и продолжить ее выполнение.

Исключения C# не поддерживают обработку асинхронных событий, таких как ошибки оборудования или прерывания, например нажатие клавиш Ctrl+C. Механизм исключений предназначен только для событий, которые могут произойти в результате работы самой программы и указываются явным образом. Исключения возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. При этом другая часть программы может попытаться сделать что-нибудь иное.

Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработка. Это важно не только для лучшей структуризации программы. Главное то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место возникновения. Это особенно актуально при использовании библиотечных функций и программ, состоящих из многих модулей.

Другое достоинство исключений состоит в том, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение или параметры, поэтому заголовки функций не разрастаются.

ПРИМЕЧАНИЕ

В принципе, ничто не мешает рассматривать в качестве исключений не только ошибки, но и нормальные ситуации, возникающие при обработке данных, но это не имеет преимуществ перед другими решениями, не улучшает структуру программы и не делает ее понятнее.

Исключения генерирует либо среда выполнения, либо программист с помощью оператора `throw`. В табл. 4.1 приведены наиболее часто используемые стандартные исключения, генерируемые средой. Они определены в пространстве имен `System`. Все они являются потомками класса `Exception`, а точнее, потомками его потомка `SystemException`.

Исключения обнаруживаются и обрабатываются в операторе `try`.

Таблица 4.1. Часто используемые стандартные исключения

Имя	Описание
`ArithmeticException`	Ошибка в арифметических операциях или преобразованиях (является предком `DivideBeZeroException` и `OverFlowException`)
`ArrayTypeMismatchException`	Попытка сохранения в массиве элемента несовместимого типа
`DivideByZeroException`	Попытка деления на ноль
`FormatException`	Попытка передать в метод аргумент неверного формата
`IndexOutOfRangeException`	Индекс массива выходит за границы диапазона
`InvalidCastException`	Ошибка преобразования типа
`OutOfMemoryException`	Недостаточно памяти для создания нового объекта
`OverFlowException`	Переполнение при выполнении арифметических операций
`StackOverFlowException`	Переполнение стека

Оператор try

Оператор `try` содержит три части:

- *контролируемый блок* — составной оператор, предваряемый ключевым словом `try`. В контролируемый блок включаются потенциально опасные операторы программы. Все функции, прямо или косвенно вызываемые из блока, также считаются ему принадлежащими;
- один или несколько *обработчиков исключений* — блоков `catch`, в которых описывается, как обрабатываются ошибки различных типов;
- *блок завершения* `finally` выполняется независимо от того, возникла ошибка в контролируемом блоке или нет.

Синтаксис оператора `try`:

```
try блок [ блоки catch ] [ блок finally ]
```

Отсутствовать могут либо блоки `catch`, либо блок `finally`, но не оба одновременно.

Рассмотрим, каким образом реализуется обработка исключительных ситуаций.

1. Обработка исключения начинается с появления ошибки. Функция или операция, в которой возникла ошибка, генерирует исключение. Как правило, исключение генерируется не непосредственно в блоке `try`, а в функциях, прямо или косвенно в него вложенных.
2. Выполнение текущего блока прекращается, отыскивается соответствующий обработчик исключения, и ему передается управление.
3. Выполняется блок `finally`, если он присутствует (этот блок выполняется и в том случае, если ошибка не возникла).

4. Если обработчик не найден, вызывается стандартный обработчик исключения. Его действия зависят от конфигурации среды. Обычно он выводит на экран окно с информацией об исключении и завершает текущий процесс.

ПРИМЕЧАНИЕ

Подобные окна не предназначены для пользователей программы, поэтому все исключения, которые могут возникнуть в программе, должны быть перехвачены и обработаны.

Обработчики исключений должны располагаться непосредственно за блоком try. Они начинаются с ключевого слова catch, за которым в скобках следует тип обрабатываемого исключения. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Блоки catch просматриваются в том порядке, в котором они записаны, пока не будет найден соответствующий типу выброшенного исключения.

Синтаксис обработчиков напоминает определение функции с одним параметром — типом исключения. Существуют три формы записи:

```
catch( тип имя ) { ... /* тело обработчика */ }
catch( тип )     { ... /* тело обработчика */ }
catch            { ... /* тело обработчика */ }
```

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий, например вывода информации об исключении.

Вторая форма не предполагает использования информации об исключении, играет роль только его тип.

Третья форма применяется для перехвата всех исключений. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа (он может быть только один) следует помещать после всех остальных. Пример:

```
try {
   ...  // Контролируемый блок
}
catch ( OverflowException e ) {
    ...    // Обработка исключений класса OverflowException (переполнение)
}
catch ( DivideByZeroException ) {
    ... // Обработка исключений класса DivideByZeroException (деление на 0)
}
catch {
    ...    // Обработка всех остальных исключений
}
```

Если исключение в контролируемом блоке не возникло, все обработчики пропускаются.

В любом случае, произошло исключение или нет, управление передается в блок завершения `finally` (если он существует), а затем — первому оператору, находящемуся непосредственно за оператором `try`. В завершающем блоке обычно записываются операторы, которые необходимо выполнить независимо от того, возникло исключение или нет, например, закрытие файлов, с которыми выполнялась работа в контролируемом блоке, или вывод информации.

В листинге 4.9 приведена программа, вычисляющая силу тока по заданным напряжению и сопротивлению. Поскольку вероятность неверного набора вещественного числа довольно высока, оператор ввода включен в контролируемый блок.

ПРИМЕЧАНИЕ

Исключение, связанное с делением на ноль, для вещественных значений возникнуть не может, поэтому не проверяется. При делении на ноль будет выдан результат «бесконечность».

Листинг 4.9. Использование исключений для проверки ввода

```
using System;
namespace ConsoleApplication1
{   class Program
    {   static void Main()
        {
            string buf;
            double u, i, r;
            try
            {
                Console.WriteLine( "Введите напряжение:" );
                u = double.Parse( Console.ReadLine() );

                Console.WriteLine( "Введите сопротивление:" );
                r = double.Parse( Console.ReadLine() );

                i = u / r;
                Console.WriteLine( "Сила тока - " + i );
            }
            catch ( FormatException )
            {
                Console.WriteLine( "Неверный формат ввода!" );
            }
            catch                                              // общий случай
            {
                Console.WriteLine( "Неопознанное исключение" );
            }
        }
    }
}
```

Операторы try могут многократно вкладываться друг в друга. Исключение, которое возникло во внутреннем блоке try и не было перехвачено соответствующим блоком catch, передается на верхний уровень, где продолжается поиск подходящего обработчика. Этот процесс называется *распространением исключения*.

Распространение исключений предоставляет программисту интересные возможности. Например, если на внутреннем уровне недостаточно информации для того, чтобы провести полную обработку ошибки, можно выполнить частичную обработку и сгенерировать исключение повторно, чтобы оно было обработано на верхнем уровне. Генерация исключения выполняется с помощью оператора throw.

Оператор throw

До сих пор мы рассматривали исключения, которые генерирует среда выполнения C#, но это может сделать и сам программист. Для генерации исключения используется оператор throw с параметром, определяющим вид исключения. Параметр должен быть объектом, порожденным от стандартного класса System.Exception. Этот объект используется для передачи информации об исключении его обработчику.

Оператор throw употребляется либо с параметром, либо без него:

```
throw [ выражение ];
```

Форма без параметра применяется только внутри блока catch для повторной генерации исключения. Тип выражения, стоящего после throw, определяет тип исключения, например:

```
throw new DivideByZeroException();
```

Здесь после слова throw записано выражение, создающее объект стандартного класса «ошибка при делении на 0» с помощью операции new.

При генерации исключения выполнение текущего блока прекращается и происходит поиск соответствующего обработчика с передачей ему управления. Обработчик считается найденным, если тип объекта, указанного после throw, либо тот же, что задан в параметре catch, либо является производным от него.

Рассмотрим пример, приведенный в спецификации C#:

```
using System;
class Test
{
    static void F() {
        try {
            G();            // функция, в которой может произойти исключение
        }
        catch ( Exception e ) {
            Console.WriteLine( "Exception in F: " + e.Message );
            e = new Exception( "F" );
            throw;                          // повторная генерация исключения
        }
```

```
    }
    static void G() {
       throw new Exception( "G" );  // моделирование исключительной ситуации
    }
    static void Main() {
       try {
           F();
       }
       catch ( Exception e ) {
           Console.WriteLine( "Exception in Main: " + e.Message );
       }
    }
}
```

В методе F выполняется промежуточная обработка исключения, которая заключается в том, что на консоль выводится поле Message перехваченного объекта e (об элементах класса Exception рассказывается в следующем разделе). После этого исключение генерируется заново. Несмотря на то что в обработчике исключения создается новый объект класса Exception с измененной строкой информации, передаваемой в исключении, выбрасывается не этот объект, а тот, который был перехвачен обработчиком, поэтому результат работы программы следующий:

```
Exception in F: G
Exception in Main: G
```

Заменим оператор throw таким оператором:

```
throw e;
```

В этом случае в обработчике будет выброшено исключение, созданное в предыдущем операторе, и вывод программы изменится:

```
Exception in F: G
Exception in Main: F
```

С обработкой исключений мы еще не раз встретимся в примерах программ, приведенных в следующих главах.

Класс Exception

Класс Exception содержит несколько полезных свойств, с помощью которых можно получить информацию об исключении. Они перечислены в табл. 4.2.

Таблица 4.2. Свойства класса System.Exception

Свойство	Описание
HelpLink	URL файла справки с описанием ошибки
Message	Текстовое описание ошибки. Устанавливается при создании объекта. Свойство доступно только для чтения
Source	Имя объекта или приложения, которое сгенерировало ошибку

Свойство	Описание
StackTrace	Последовательность вызовов, которые привели к возникновению ошибки. Свойство доступно только для чтения
InnerException	Содержит ссылку на исключение, послужившее причиной генерации текущего исключения
TargetSite	Метод, выбросивший исключение

Операторы checked и unchecked

Как уже упоминалось в главе 3, процессом генерации исключений, возникающих при переполнении, можно управлять с помощью ключевых слов checked и unchecked, которые употребляются как операции, если они используются в выражениях, и как операторы, если они предваряют блок, например:

```
a = checked (b + c);          // для выражения (проверка включена)
unchecked {                   // для блока операторов (проверка выключена)
   a = b + c;
}
```

Проверка не распространяется на функции, вызванные в блоке.

Рекомендации по программированию

К настоящему моменту вы изучили основные операторы C# и можете писать на этом языке простые программы, используя класс, состоящий из одного метода. Даже на этом уровне важно придерживаться определенных правил, следование которым поможет вам избежать многих распространенных ошибок. Конечно, на все случаи жизни советы дать невозможно, ведь не зря многие считают программирование искусством. С приобретением опыта вы добавите к приведенным далее правилам множество своих, выстраданных в результате долгих бдений над «последней» ошибкой.

ВНИМАНИЕ

Главная цель, к которой нужно стремиться, — получить понятную программу как можно более простой структуры. В конечном счете, все технологии программирования направлены на достижение именно этой цели, поскольку только так можно добиться надежности программы и легкости ее модификации.

Создание программы надо начинать с определения ее *исходных данных и результатов*. При этом задумываются не только о смысле каждой величины, но и о том, какое множество значений она может принимать. В соответствии с этим выбираются типы переменных.

Следующий шаг — *записать на естественном языке* (возможно, с применением обобщенных блок-схем), что именно и как должна делать программа. Если вы не

можете сформулировать алгоритм по-русски, велика вероятность того, что он плохо продуман (естественно, я не имею в виду, что надо «проговаривать» все на уровне отдельных операторов, например, «изменяя индекс от 0 до 100 с шагом 1...»). Описание алгоритма полезно по нескольким причинам: оно помогает в деталях продумать алгоритм, найти на самой ранней стадии некоторые ошибки, разбить программу на логическую последовательность блоков, а также обеспечить комментарии к программе.

При кодировании[1] программы необходимо помнить о принципах *структурного программирования:* программа должна состоять из четкой последовательности блоков — базовых конструкций, каждая из которых имеет один вход и один выход. Конструкции могут вкладываться, но не пересекаться.

Программа должна быть «прозрачна». Если какое-либо действие можно запрограммировать разными способами, то предпочтение должно отдаваться не наиболее компактному и даже не наиболее эффективному, а более понятному. Особенно это важно тогда, когда пишут программу одни, а сопровождают другие, что является широко распространенной практикой. «Непрозрачное» программирование может повлечь за собой огромные издержки, связанные с поиском ошибок при отладке.

Для записи каждого фрагмента алгоритма необходимо использовать наиболее подходящие средства языка. Например, ветвление на насколько направлений по значению целой или строковой переменной эффектнее записать с помощью одного оператора `switch`, а не нескольких операторов `if`. Для просмотра массива лучше пользоваться циклом `for` или `foreach`. Оператор `goto` применяют весьма редко, например, в операторе выбора `switch` или для принудительного выхода из нескольких вложенных циклов, а в большинстве других ситуаций лучше использовать другие средства языка, такие как `break` или `return`.

Для организации циклов пользуйтесь наиболее подходящим оператором. Цикл `do` применяется только в тех случаях, когда тело в любом случае потребуется выполнить хотя бы один раз, например, при проверке ввода. При использовании циклов надо стремиться объединять инициализацию, проверку условия выхода и приращение в одном месте. Рекомендации по выбору наиболее подходящего оператора цикла были приведены на с. 82.

При записи итеративных циклов (в которых для проверки условия выхода используются соотношения переменных, формирующихся в теле цикла), необходимо *предусматривать аварийный выход* по достижении заранее заданного максимального количества итераций. Это повышает надежность программы.

Более короткую ветвь if лучше поместить сначала, иначе вся структура может не поместиться на экране, что затруднит отладку.

Бессмысленно использовать проверку на равенство `true` *или* `false`:

```
bool busy;
...
if ( busy == true )  { ... }      // плохо! Лучше if ( busy )
if ( busy == false ) { ... }      // плохо! Лучше if ( !busy )
```

[1] *Кодированием* называют процесс написания текста программы, чтобы отличить его от более общего понятия «программирование», включающего также этапы проектирования и отладки программы.

Следует избегать лишних проверок условий. Например:

```
if ( a < b ) c = 1;
else if ( a > b ) c = 2;
     else if ( a == b ) c = 3;
```

Вместо этих операторов можно написать:

```
if      ( a < b ) c = 1;
else if ( a > b ) c = 2;
else              c = 3;
```

Или даже так:

```
c = 3;
if ( a < b ) c = 1;
if ( a > b ) c = 2;
```

Если первая ветвь оператора `if` обеспечивает передачу управления, *использовать ветвь else нет необходимости*:

```
if ( i > 0 ) break;
   // здесь i <= 0
```

В некоторых случаях *условная операция лучше условного оператора*:

```
if ( z == 0 ) i = j; else i = k;     // лучше так: i = z == 0 ? j : k;
```

Необходимо предусматривать печать сообщений или генерацию исключения в тех точках программы, куда управление при нормальной работе программы передаваться не должно. Именно это сообщение вы с большой вероятностью получите при первом же запуске программы. Например, полезно, если оператор `switch` имеет ветвь `default`, реализующую обработку ситуации по умолчанию, если в ней перечислены все возможные значения переключателя.

В программе полезно предусматривать реакцию на неверные входные параметры. Это может быть генерация исключения (предпочтительно), печать сообщения или формирование признака результата с его последующим анализом. *Сообщение об ошибке должно быть информативным* и подсказывать пользователю, как ее исправить. Например, при вводе неверного значения в сообщении должен быть указан допустимый диапазон.

Генерируйте исключения в тех случаях, когда в месте возникновения ошибки недостаточно данных, чтобы ее обработать.

Заключайте потенциально опасные фрагменты программы в *проверяемый блок* `try` и обрабатывайте хотя бы исключение типа `Exception`, а лучше — все исключения, которые могут в нем возникнуть, по отдельности.

Конечно, все усилия по комментированию программы пропадут втуне, если сама она написана путано, неряшливо и непродуманно. Поэтому *после написания программу следует тщательно отредактировать* — убрать ненужные фрагменты, сгруппировать описания, оптимизировать проверки условий и циклы, проверить, оптимально ли разбиение на методы, и т. д. С первого раза без помарок хороший текст не напишешь, будь то сочинение, статья или программа. Однако не следует

пытаться оптимизировать все, что «попадается под руку», поскольку *главный принцип программиста тот же, что и врача: «Не навреди!»*. Если программа работает недостаточно эффективно, надо в первую очередь подумать о том, какие алгоритмы в нее заложены.

Подходить к написанию программы нужно таким образом, *чтобы ее можно было в любой момент передать другому программисту*. Полезно дать почитать свою программу кому-нибудь из друзей или коллег (а еще лучше — врагов или завистников) и в тех местах, которые они не смогут понять без устных комментариев, внести их прямо в текст программы.

Комментарии имеют очень важное значение, поскольку программист, как ни странно, чаще читатель, чем писатель. Даже если сопровождающим программистом является автор программы, разбираться через год в плохо документированном тексте — сомнительное удовольствие. Дальнейшие советы касаются комментариев и форматирования текста программы, что является неотъемлемой частью процесса программирования.

Программа, если она используется, живет не один год, потребность в каких-то ее новых свойствах проявляется сразу же после ввода в эксплуатацию, и сопровождение программы занимает гораздо больше времени, чем ее написание. Основная часть документации должна находиться в тексте программы. Хорошие комментарии написать почти так же сложно, как и хорошую программу. В C# есть мощные средства документирования программ, речь о которых пойдет в главе 15, но само содержание комментариев никто, кроме вас, задать не сможет.

Комментарии должны представлять собой правильные предложения без сокращений и со знаками препинания[1] и не должны подтверждать очевидное (комментарии в этой книге не могут служить образцом, поскольку они предназначены для обучения, а не сопровождения). Например, бессмысленны фразы типа «вызов метода f» или «описание переменных». Если комментарий к фрагменту программы занимает несколько строк, разместить его лучше перед фрагментом, чем справа от него, и выровнять по вертикали.

Вложенные блоки должны иметь отступ в 3–5 символов, причем блоки одного уровня вложенности должны быть выровнены по вертикали. *Форматируйте текст по столбцам* везде, где это возможно, это делает программу гораздо понятнее:

```
    string buf    = "qwerty";
    double ex     = 3.1234;
    int    number = 12;
    byte   z      = 0;
...
    if ( done ) Console.WriteLine( "Сумма ряда - " + y );
    else        Console.WriteLine( "Ряд расходится" );
    ...
    if      ( x >= 0 && x < 10 ) y = t * x;
    else if ( x >= 10 )          y = 2 * t;
    else                         y = x;
```

[1] Совсем хорошо, если они при этом не будут содержать орфографических ошибок.

В последних трех строках показано, что иногда большей ясности можно добиться, если не следовать правилу отступов буквально.

Абзацный отступ комментария должен соответствовать отступу комментируемого блока:

```
// Комментарий, описывающий,
// что происходит в следующем ниже
// блоке программы.
{ /* Непонятный
  блок
  программы */
}
```

Для разделения методов и других логически законченных фрагментов пользуйтесь пустыми строками или комментарием вида

```
// ------------------------------------------------------
```

Не следует размещать в одной строке много операторов. Как и в русском языке, *после знаков препинания должны использоваться пробелы*:

```
f=a+b;                    // плохо! Лучше f = a + b;
```

Помечайте конец длинного составного оператора, например:

```
while ( true )
{
    while ( x < y )
    {
        for ( i = 0; i < 10; ++i )
        {
            for ( j = 0; j < 10; ++j )
            {
                // две страницы кода
            }                         // end for ( j = 0; j < 10; ++j )
        }                             // end for ( i = 0; i < 10; ++i )
    }                                 // end while ( x < y )
}                                     // end while ( true )
```

Конструкции языка C# в основном способствуют хорошему стилю программирования, тем не менее, и на этом языке легче легкого написать запутанную, ненадежную, некрасивую программу, которую проще переписать заново, чем внести в нее требуемые изменения. Только постоянные тренировки, самоконтроль и стремление к совершенствованию помогут вам освоить хороший стиль, без которого невозможно стать квалифицированным программистом[1].

В заключение порекомендую тем, кто предпочитает учиться программированию не только на своих ошибках, очень полезные книги [2], [6].

---

[1] Вообще говоря, этот не блещущий новизной совет можно отнести практически к любому роду человеческой деятельности.

Глава 5
Классы: основные понятия

Понятие о классах вы получили[1] в разделах «Классы» (см. с. 13) и «Заготовка консольной программы» (см. с. 17). Все программы, приведенные в этой книге ранее, состояли из одного класса с одним-единственным методом `Main`. Сейчас настало время подробнее изучить состав, правила создания и использования классов. По сути, отныне все, что мы будем рассматривать, так или иначе связано с этим ключевым средством языка.

Класс является типом данных, определяемым пользователем. Он должен представлять собой одну логическую сущность, например, являться моделью реального объекта или процесса. *Элементами* класса являются *данные* и *функции*, предназначенные для их обработки.

Описание класса содержит ключевое слово `class`, за которым следует его *имя*, а далее в фигурных скобках — *тело* класса, то есть список его элементов. Кроме того, для класса можно задать его базовые классы (предки) и ряд необязательных атрибутов и спецификаторов, определяющих различные характеристики класса:

```
[ атрибуты ] [ спецификаторы ] class имя_класса [ : предки ]
    тело_класса
```

Как видите, обязательными являются только ключевое слово `class`, а также имя и тело класса. *Имя класса* задается программистом по общим правилам C#. *Тело класса* — это список описаний его элементов, заключенный в фигурные скобки. Список может быть пустым, если класс не содержит ни одного элемента. Таким образом, простейшее описание класса может выглядеть так:

```
class Demo {}
```

ПРИМЕЧАНИЕ

Необязательные *атрибуты* задают дополнительную информацию о классе. Поскольку наша задача пока состоит в том, чтобы освоить основные понятия, мы отложим знакомство с атрибутами до главы 12.

[1] А может быть, и не получили.

Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечислены в табл. 5.1. Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется *вложенным*. В зависимости от места описания класса некоторые из этих спецификаторов могут быть запрещены.

Таблица 5.1. Спецификаторы класса

№	Спецификатор	Описание
1	`new`	Используется для вложенных классов. Задает новое описание класса взамен унаследованного от предка. Применяется в иерархиях объектов, рассматривается в главе 8 (см. с. 175)
2	`public`	Доступ не ограничен
3	`protected`	Используется для вложенных классов. Доступ только из элементов данного и производных классов
4	`internal`	Доступ только из данной программы (сборки)[1]
5	`protected internal`	Доступ только из данного и производных классов или из данной программы (сборки)
6	`private`	Используется для вложенных классов. Доступ только из элементов класса, внутри которого описан данный класс
7	`abstract`	Абстрактный класс. Применяется в иерархиях объектов, рассматривается в главе 8 (см. с. 181)
8	`sealed`	Бесплодный класс. Применяется в иерархиях объектов, рассматривается в главе 8 (см. с. 182)
9	`static`	Статический класс. Введен в версию языка 2.0. Рассматривается в разделе «Конструкторы» (см. с. 114)

Спецификаторы 2–6 называются *спецификаторами доступа*. Они определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут присутствовать в описании только в вариантах, приведенных в таблице, а также могут комбинироваться с остальными спецификаторами.

В этой главе мы будем изучать классы, которые описываются в пространстве имен непосредственно (то есть не вложенные классы). Для таких классов допускаются только два спецификатора: `public` и `internal`. По умолчанию, то есть если ни один спецификатор доступа не указан, подразумевается спецификатор `internal`.

Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых *экземплярами*, или *объектами*, *класса*.

[1] Понятие сборки было введено в главе 1 на с. 9.

Объекты создаются явным или неявным образом, то есть либо программистом, либо системой. Программист создает экземпляр класса с помощью операции new, например:

```
Demo a = new Demo();        // создание экземпляра класса Demo
Demo b = new Demo();        // создание другого экземпляра класса Demo
```

ПРИМЕЧАНИЕ

Как вы помните, класс относится к ссылочным типам данных, память под которые выделяется в хипе (см. раздел «Типы-значения и ссылочные типы» на с. 35). Таким образом, переменные *x* и *y* хранят не сами объекты, а ссылки на объекты, то есть их адреса. Если достаточный для хранения объекта объем памяти выделить не удалось, операция new генерирует исключение OutOfMemoryException. Рекомендуется предусматривать обработку этого исключения в программах, работающих с объектами большого объема.

Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные. Кроме того, в классе могут присутствовать *статические элементы*, которые существуют в единственном экземпляре для всех объектов класса. Часто статические данные называют *данными класса*, а остальные — *данными экземпляра*.

Функциональные элементы класса не тиражируются, то есть всегда хранятся в единственном экземпляре. Для работы с данными класса используются *методы класса* (*статические методы*), для работы с данными экземпляра — *методы экземпляра*, или просто *методы*.

До сих пор мы использовали в программах только один вид функциональных элементов класса — методы. Поля и методы являются основными элементами класса. Кроме того, в классе можно задавать целую гамму других элементов: свойства, события, индексаторы, операции, конструкторы, деструкторы, а также типы (рис. 5.1).

Ниже приведено краткое описание всех элементов класса (см. также рис. 5.1):

- *Константы* класса хранят неизменяемые значения, связанные с классом.
- *Поля* содержат данные класса.
- *Методы* реализуют вычисления или другие действия, выполняемые классом или экземпляром.
- *Свойства* определяют характеристики класса в совокупности со способами их задания и получения, то есть методами записи и чтения.
- *Конструкторы* реализуют действия по инициализации экземпляров или класса в целом.
- *Деструкторы* определяют действия, которые необходимо выполнить до того, как объект будет уничтожен.
- *Индексаторы* обеспечивают возможность доступа к элементам класса по их порядковому номеру.

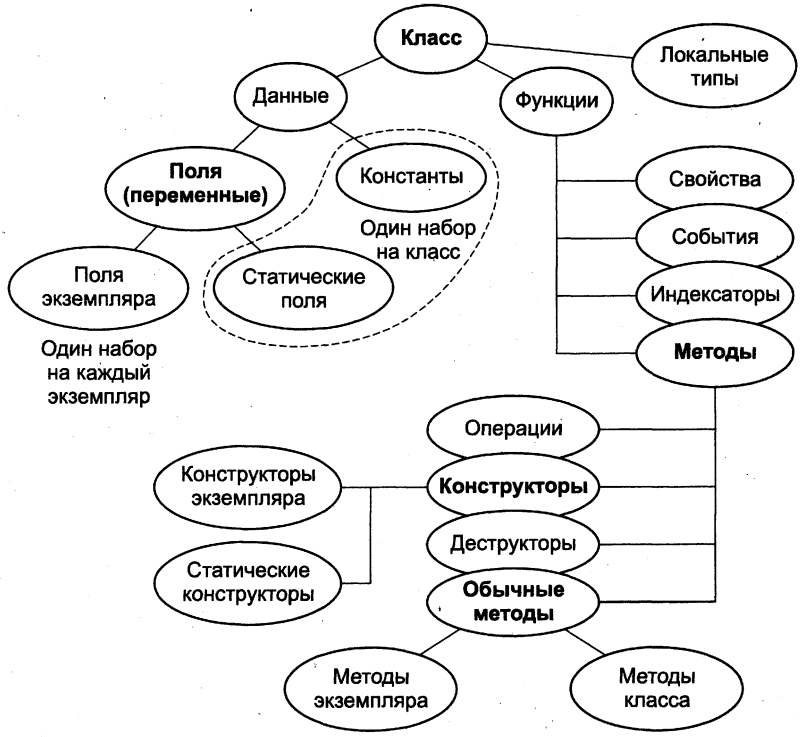


Рис. 5.1. Состав класса

- *Операции* задают действия с объектами с помощью знаков операций.
- *События* определяют уведомления, которые может генерировать класс.
- *Типы* — это типы данных, внутренние по отношению к классу.

Первые пять видов элементов класса мы рассмотрим в этой главе, а остальные — в последующих. Но прежде чем начать изучение, необходимо поговорить о присваивании и сравнении объектов.

Присваивание и сравнение объектов

Операция присваивания рассматривалась в разделе «Операции присваивания» (см. с. 57). Механизм выполнения присваивания один и тот же для величин любого типа, как ссылочного, так и значимого, однако результаты различаются. При присваивании значения копируется значение, а при присваивании ссылки — ссылка, поэтому после присваивания одного объекта другому мы получим две ссылки, указывающие на одну и ту же область памяти (рис. 5.2).

Рисунок иллюстрирует ситуацию, когда было создано три объекта, `a`, `b` и `c`, а затем выполнено присваивание `b = c`. Старое значение `b` становится недоступным и очищается сборщиком мусора. Из этого следует, что если изменить значение

одной величины ссылочного типа, это может отразиться на другой (в данном случае, если изменить объект через ссылку c, объект b также изменит свое значение).

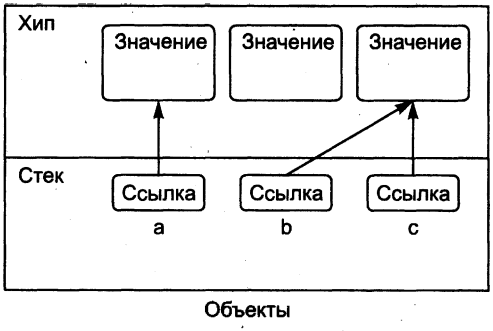


Рис. 5.2. Присваивание объектов

Аналогичная ситуация с операцией проверки на равенство. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке объекты b и c равны, но a не равно b даже при равенстве их значений или если они обе равны null).

Данные: поля и константы

Данные, содержащиеся в классе, могут быть *переменными* или *константами* и задаются в соответствии с правилами, рассмотренными в разделе «Переменные» (см. с. 38) и «Именованные константы» (см. с. 41). Переменные, описанные в классе, называются *полями* класса.

При описании элементов класса можно также указывать атрибуты и спецификаторы, задающие различные характеристики элементов. Синтаксис описания элемента данных приведен ниже:

```
[ атрибуты ] [ спецификаторы ] [ const ] тип имя [ = начальное_значение ]
```

До *атрибутов* мы доберемся еще не скоро, в главе 12, а возможные *спецификаторы* полей и констант перечислены в табл. 5.2. Для констант можно использовать только спецификаторы 1–6.

Таблица 5.2. Спецификаторы полей и констант класса

№	Спецификатор	Описание
1	new	Новое описание поля, скрывающее унаследованный элемент класса
2	public	Доступ к элементу не ограничен
3	protected	Доступ только из данного и производных классов
4	internal	Доступ только из данной сборки

№	Спецификатор	Описание
5	protected internal	Доступ только из данного и производных классов и из данной сборки
6	private	Доступ только из данного класса
7	static	Одно поле для всех экземпляров класса
8	readonly	Поле доступно только для чтения
9	volatile	Поле может изменяться другим процессом или системой

По умолчанию элементы класса считаются закрытыми (private). Для полей класса этот вид доступа является предпочтительным, поскольку поля определяют внутреннее строение класса, которое должно быть скрыто от пользователя. Все методы класса имеют непосредственный доступ к его закрытым полям.

ВНИМАНИЕ

Поля, описанные со спецификатором static, а также константы существуют в единственном экземпляре для всех объектов класса, поэтому к ним обращаются не через имя экземпляра, а через имя класса. Если класс содержит только статические элементы, экземпляр класса создавать не требуется. Именно этим фактом мы пользовались во всех предыдущих листингах.

Обращение к полю класса выполняется с помощью *операции доступа* (точка). Справа от точки задается имя поля, слева — имя экземпляра для обычных полей или имя класса для статических. В листинге 5.1 приведены пример простого класса Demo и два способа обращения к его полям.

Листинг 5.1. Класс Demo, содержащий поля и константу

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
        public int a = 1;                  // поле данных
        public const double  c = 1.66;     // константа
        public static string s = "Demo";   // статическое поле класса
        double y;                          // закрытое поле данных
    }

    class Class1
    {   static void Main()
        {
            Demo x = new Demo();           // создание экземпляра класса Demo
            Console.WriteLine( x.a );      // x.a - обращение к полю класса
            Console.WriteLine( Demo.c );   // Demo.c - обращение к константе
            Console.WriteLine( Demo.s );   // обращение к статическому полю
        }
    }
}
```

Поле y вывести на экран аналогичным образом не удастся: оно является закрытым, то есть недоступно извне (из класса Class1). Поскольку значение этому полю явным образом не присвоено, среда присваивает ему значение ноль.

ВНИМАНИЕ

Все поля сначала автоматически инициализируются нулем соответствующего типа (например, полям типа int присваивается 0, а ссылкам на объекты — значение null). После этого полю присваивается значение, заданное при его явной инициализации. Задание начальных значений для статических полей выполняется при инициализации класса, а обычных — при создании экземпляра.

Поля со спецификатором readonly предназначены только для чтения. Установить значение такого поля можно либо при его описании, либо в конструкторе (конструкторы рассматриваются далее в этой главе).

Методы

Метод — это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или экземпляром. Методы определяют поведение класса.

Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может столько раз, сколько необходимо. Один и тот же метод может обрабатывать различные данные, переданные ему в качестве аргументов.

Синтаксис метода:

```
[ атрибуты ] [ спецификаторы ] тип имя_метода ( [ параметры ] )
    тело_метода
```

Рассмотрим основные элементы описания метода. Первая строка представляет собой *заголовок* метода. *Тело метода*, задающее действия, выполняемые методом, чаще всего представляет собой блок — последовательность операторов в фигурных скобках[1].

При описании методов можно использовать спецификаторы 1–7 из табл. 5.2, имеющие тот же смысл, что и для полей, а также спецификаторы virtual, sealed, override, abstract и extern, которые будут рассмотрены по мере необходимости. Чаще всего для методов задается спецификатор доступа public, ведь методы составляют интерфейс класса — то, с чем работает пользователь, поэтому они должны быть доступны.

ВНИМАНИЕ

Статические (static) методы, или методы класса, можно вызывать, не создавая экземпляр объекта. Именно таким образом используется метод Main.

[1] Позже мы увидим, что для методов некоторых типов вместо тела используется просто точка с запятой.

Пример простейшего метода:

```
public double Gety()            // метод для получения поля y из листинга 5.1
   {
      return y;
   }
```

Тип определяет, значение какого типа вычисляется с помощью метода. Часто употребляется термин «метод возвращает значение», поскольку после выполнения метода происходит возврат в то место вызывающей функции, откуда был вызван метод, и передача туда значения выражения, записанного в операторе return (рис. 5.3). Если метод не возвращает никакого значения, в его заголовке задается тип void, а оператор return отсутствует.

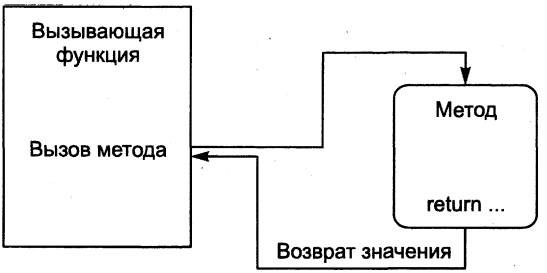


Рис. 5.3. Вызов метода

Параметры используются для обмена информацией с методом. Параметр представляет собой локальную переменную, которая при вызове метода принимает значение соответствующего аргумента. Область действия параметра — весь метод.

Например, чтобы вычислить значение синуса для вещественной величины x, мы передаем ее в качестве аргумента в метод Sin класса Math, а чтобы вывести значение этой переменной на экран, мы передаем ее в метод WriteLine класса Console:

```
double x = 0.1;
double y = Math.Sin(x);
Console.WriteLine(x);
```

При этом метод Sin возвращает в точку своего вызова вещественное значение синуса, которое присваивается переменной y, а метод WriteLine ничего не возвращает.

ВНИМАНИЕ

Метод, не возвращающий значение, вызывается отдельным оператором, а метод, возвращающий значение, — в составе выражения в правой части оператора присваивания.

Параметры, описываемые в заголовке метода, определяют множество значений *аргументов*, которые можно передавать в метод. Список аргументов при вызове как бы накладывается на список параметров, поэтому они должны попарно

соответствовать друг другу. Правила соответствия подробно рассматриваются в следующих разделах.

Для каждого параметра должны задаваться его *тип* и *имя*. Например, заголовок метода Sin выглядит следующим образом:

```
public static double Sin( double a );
```

Имя метода вкупе с количеством, типами и спецификаторами его параметров представляет собой *сигнатуру метода* — то, по чему один метод отличают от других. В классе не должно быть методов с одинаковыми сигнатурами.

В листинге 5.2 в класс Demo добавлены методы установки и получения значения поля y (на самом деле для подобных целей используются не методы, а свойства, которые рассматриваются чуть позже). Кроме того, статическое поле s закрыто, то есть определено по умолчанию как private, а для его получения описан метод Gets, являющий собою пример статического метода.

Листинг 5.2. Простейшие методы

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
        public int a = 1;
        public const double c = 1.66;
        static string s = "Demo";
        double y;

        public double Gety()                    // метод получения поля y
        {
            return y;
        }

        public void Sety( double y_ )           // метод установки поля y
        {
            y = y_;
        }

        public static string Gets()             // метод получения поля s
        {
            return s;
        }
    }

    class Class1
    {   static void Main()
        {
            Demo x = new Demo();
            x.Sety(0.12);                       // вызов метода установки поля y
```

```
            Console.WriteLine( x.Gety() );    // вызов метода получения поля y
            Console.WriteLine( Demo.Gets() ); // вызов метода получения поля s
//          Console.WriteLine( Gets() );      // при вызове из др. метода этого объекта
        }
    }
}
```

Как видите, методы класса имеют непосредственный доступ к его закрытым полям. Метод, описанный со спецификатором `static`, должен обращаться только к статическим полям класса. Обратите внимание на то, что *статический метод вызывается через имя класса, а обычный — через имя экземпляра.*

ПРИМЕЧАНИЕ

При вызове метода из другого метода того же класса имя класса/экземпляра можно не указывать.

Параметры методов

Рассмотрим более подробно, каким образом метод обменивается информацией с вызвавшим его кодом. При вызове метода выполняются следующие действия:

1. Вычисляются выражения, стоящие на месте аргументов.
2. Выделяется память под параметры метода в соответствии с их типом.
3. Каждому из параметров сопоставляется соответствующий аргумент (аргументы как бы накладываются на параметры и замещают их).
4. Выполняется тело метода.
5. Если метод возвращает значение, оно передается в точку вызова; если метод имеет тип `void`, управление передается на оператор, следующий после вызова.

При этом проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение. Листинг 5.3 иллюстрирует этот процесс.

Листинг 5.3. Передача параметров методу

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static int Max(int a, int b)  // метод выбора максимального значения
        {
            if ( a > b ) return a;
            else         return b;
        }
        static void Main()
        {
            int a = 2, b = 4;
            int x = Max( a, b );                  // вызов метода Max
            Console.WriteLine( x );               // результат: 4
```

продолжение

Листинг 5.3 (*продолжение*)

```
            short t1 = 3, t2 = 4;
            int y = Max( t1, t2 );              // вызов метода Max
            Console.WriteLine( y );             // результат: 4

            int z = Max( a + t1, t1 / 2 * b );  // вызов метода Max
            Console.WriteLine( z );             // результат: 5
        }
    }
}
```

В классе описан метод `Max`, который выбирает наибольшее из двух переданных ему значений. Параметры описаны как `a` и `b`. В методе `Main` выполняются три вызова `Max`. В результате первого вызова методу `Max` передаются два аргумента того же типа, что и параметры, во втором вызове — аргументы совместимого типа, в третьем — выражения.

ВНИМАНИЕ

Главное требование при передаче параметров состоит в том, что аргументы при вызове метода должны записываться в том же порядке, что и параметры в заголовке метода. Правила соответствия типов аргументов и параметров описаны в следующих разделах.

Количество аргументов должно соответствовать количеству параметров. На имена никаких ограничений не накладывается: имена аргументов могут как совпадать, так и не совпадать с именами параметров.

Существуют два способа передачи параметров: по значению и по ссылке.

При передаче по значению метод получает копии значений аргументов, и операторы метода работают с этими копиями. Доступа к исходным значениям аргументов у метода нет, а следовательно, нет и возможности их изменить.

При передаче по ссылке (*по адресу*) метод получает копии адресов аргументов, он осуществляет доступ к ячейкам памяти по этим адресам и может изменять исходные значения аргументов, модифицируя параметры.

В C# для обмена данными между вызывающей и вызываемой функциями предусмотрено четыре типа параметров:

- параметры-значения;
- параметры-ссылки — описываются с помощью ключевого слова `ref`;
- выходные параметры — описываются с помощью ключевого слова `out`;
- параметры-массивы — описываются с помощью ключевого слова `params`.

Ключевое слово предшествует описанию типа параметра. Если оно опущено, параметр считается параметром-значением. Параметр-массив может быть только один и должен располагаться последним в списке, например:

```
public int Calculate( int a, ref int b, out int c, params int[] d ) …
```

О параметрах-массивах мы будем говорить позже, в главе 7 (см. с. 154), а сейчас рассмотрим остальные типы параметров.

Параметры-значения

Параметр-значение описывается в заголовке метода следующим образом:

```
тип имя
```

Пример заголовка метода, имеющего один параметр-значение целого типа:

```
void P( int x )
```

Имя параметра может быть произвольным. Параметр x представляет собой локальную переменную, которая получает свое значение из вызывающей функции при вызове метода. В метод передается копия значения аргумента.

Механизм передачи следующий: из ячейки памяти, в которой хранится переменная, передаваемая в метод, берется ее значение и копируется в специальную область памяти — область параметров. Метод работает с этой копией, следовательно, доступа к ячейке, где хранится сама переменная, не имеет. По завершении работы метода область параметров освобождается. Таким образом, для параметров-значений используется, как вы догадались, *передача по значению*. Ясно, что этот способ годится только для величин, которые не должны измениться после выполнения метода, то есть для его исходных данных.

При вызове метода на месте параметра, передаваемого по значению, может находиться выражение, а также, конечно, его частные случаи — переменная или константа. Должно существовать неявное преобразование типа выражения к типу параметра[1].

Например, пусть в вызывающей функции описаны переменные и им до вызова метода присвоены значения:

```
int    x = 1;
sbyte  c = 1;
ushort y = 1;
```

Тогда следующие вызовы метода P, заголовок которого был описан ранее, будут синтаксически правильными:

```
P( x );    P( c );    P( y );    P( 200 );    P( x / 4 + 1 );
```

Параметры-ссылки

Во многих методах все величины, которые метод должен получить в качестве исходных данных, описываются в списке параметров, а величина, которую вычисляет метод как результат своей работы, возвращается в вызывающий код с помощью оператора return. Очевидно, что если метод должен возвращать более одной величины, такой способ не годится. Еще одна проблема возникает, если в методе

[1] О неявных преобразованиях рассказывалось в разделе «Преобразования встроенных арифметических типов-значений» (см. с. 45).

требуется изменить значение каких-либо передаваемых в него величин. В этих случаях используются *параметры-ссылки*.

Признаком параметра-ссылки является ключевое слово ref перед описанием параметра:

```
ref тип имя
```

Пример заголовка метода, имеющего один параметр-ссылку целого типа:

```
void P( ref int x )
```

При вызове метода в область параметров копируется не значение аргумента, а его адрес, и метод через него имеет доступ к ячейке, в которой хранится аргумент. Таким образом, параметры-ссылки передаются *по адресу* (чаще употребляется термин «передача по ссылке»). Метод работает непосредственно с переменной из вызывающей функции и, следовательно, может ее изменить, поэтому если в методе требуется изменить значения параметров, они должны передаваться только по ссылке.

ВНИМАНИЕ

При вызове метода на месте параметра-ссылки может находиться только ссылка на инициализированную переменную точно того же типа. Перед именем параметра указывается ключевое слово ref.

Исходные данные передавать в метод по ссылке не рекомендуется, чтобы исключить возможность их непреднамеренного изменения.

Проиллюстрируем передачу параметров-значений и параметров-ссылок на примере (листинг 5.4).

Листинг 5.4. Параметры-значения и параметры-ссылки

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void P( int a, ref int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
            int a = 2, b = 4;
            Console.WriteLine( "до вызова     {0} {1}", a, b );
            P( a, ref b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
    }
}
```

Результаты работы этой программы:

```
до вызова      2 4
внутри метода 44 33
после вызова   2 33
```

Как видите, значение переменной a в функции Main не изменилось, поскольку переменная передавалась по значению, а значение переменной b изменилось потому, что она была передана по ссылке.

Несколько иная картина получится, если передавать в метод не величины значимых типов, а *экземпляры классов*, то есть величины ссылочных типов. Как вы помните, переменная-объект на самом деле хранит ссылку на данные, расположенные в динамической памяти, и именно эта ссылка передается в метод либо по адресу, либо по значению. В обоих случаях метод получает в свое распоряжение фактический адрес данных и, следовательно, может их изменить.

СОВЕТ

Для простоты можно считать, что объекты всегда передаются по ссылке.

Разница между передачей объектов по значению и по ссылке состоит в том, что в последнем случае можно изменить саму ссылку, то есть после вызова метода она может указывать на другой объект.

Выходные параметры

Довольно часто возникает необходимость в методах, которые формируют несколько величин, например, если в методе создаются объекты или инициализируются ресурсы. В этом случае становится неудобным ограничение параметров-ссылок: необходимость присваивания значения аргументу до вызова метода. Это ограничение снимает спецификатор out. Параметру, имеющему этот спецификатор, должно быть обязательно присвоено значение внутри метода, компилятор за этим следит. Зато в вызывающем коде можно ограничиться описанием переменной без инициализации.

Изменим описание второго параметра в листинге 5.4 так, чтобы он стал *выходным* (листинг 5.5).

Листинг 5.5. Выходные параметры

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void P( int a, out int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
```

продолжение

Листинг 5.5 (*продолжение*)

```
            int a = 2, b;
            P( a, out b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
    }
}
```

При вызове метода перед соответствующим параметром тоже указывается ключевое слово out.

СОВЕТ

В списке параметров записывайте сначала все входные параметры, затем — все ссылки и выходные параметры. Давайте параметрам имена, по которым можно получить представление об их назначении.

Ключевое слово this

Каждый объект содержит свой экземпляр полей класса. Методы находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший функцию экземпляр.

В явном виде параметр this применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

```
class Demo
{
    double y;
    public Demo T()          // метод T возвращает ссылку на экземпляр
    {
        return this;
    }
    public void Sety( double y )
    {
        this.y = y;          // полю y присваивается значение параметра y
    }
}
```

Конструкторы

Конструктор предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора совпадает с именем класса. Ниже перечислены свойства конструкторов:

- Конструктор *не возвращает значение*, даже типа void.

- Класс может иметь *несколько конструкторов* с разными параметрами для разных видов инициализации.
- Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается нуль, полям ссылочных типов — значение null.
- Конструктор, вызываемый без параметров, называется *конструктором по умолчанию*.

До сих пор мы задавали начальные значения полей класса при описании класса (см., например, листинг 5.1). Это удобно в том случае, когда для всех экземпляров класса начальные значения некоторого поля одинаковы. Если же при создании объектов требуется присваивать полю разные значения, это следует делать в конструкторе. В листинге 5.6 в класс Demo добавлен конструктор, а поля сделаны закрытыми (ненужные в данный момент элементы опущены). В программе создаются два объекта с различными значениями полей.

Листинг 5.6. Класс с конструктором

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
        public Demo( int a, double y )          // конструктор с параметрами
        {
            this.a = a;
            this.y = y;
        }

        public double Gety()                    // метод получения поля y
        {
            return y;
        }

        int a;
        double y;
    }

    class Class1
    {   static void Main()
        {
            Demo a = new Demo( 300, 0.002 );    // вызов конструктора
            Console.WriteLine( a.Gety() );      // результат: 0,002
            Demo b = new Demo( 1, 5.71 );       // вызов конструктора
            Console.WriteLine( b.Gety() );      // результат: 5,71
        }
    }
}
```

Часто бывает удобно задать в классе *несколько конструкторов*, чтобы обеспечить возможность инициализации объектов разными способами. Следующий пример несколько «притянут за уши», но тем не менее иллюстрирует этот тезис:

```
class Demo
{
    public Demo( int a )                    // конструктор 1
    {
        this.a = a;
        this.y = 0.002;
    }

    public Demo( double y )                 // конструктор 2
    {
        this.a = 1;
        this.y = y;
    }
    ...
}
    ...
        Demo x = new Demo( 300 );           // вызов конструктора 1
        Demo y = new Demo( 5.71 );          // вызов конструктора 2
```

Все конструкторы должны иметь разные сигнатуры.

Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-нибудь, удобно *вызвать первый конструктор из второго*. Для этого используется уже известное вам ключевое слово this в другом контексте, например:

```
class Demo
{
    public Demo( int a )                          // конструктор 1
    {
        this.a = a;
    }
    public Demo( int a, double y ) : this( a )    // вызов конструктора 1
    {
        this.y = y;
    }
    ...
}
```

Конструкция, находящаяся после двоеточия, называется *инициализатором*, то есть тем кодом, который исполняется до начала выполнения тела конструктора.

Как вы помните, все классы в C# имеют общего предка — класс object. Конструктор любого класса, если не указан инициализатор, автоматически вызывает конструктор своего предка. Это можно сделать и явным образом с помощью

ключевого слова base, обозначающего конструктор базового класса. Таким образом, первый конструктор из предыдущего примера можно записать и так:

```
public Demo( int a ) : base()              // конструктор 1
{
    this.a = a;
}
```

ПРИМЕЧАНИЕ

Конструктор базового класса вызывается явным образом в тех случаях, когда ему требуется передать параметры.

До сих пор речь шла об «обычных» конструкторах, или *конструкторах экземпляра*. Существует второй тип конструкторов — *статические конструкторы*, или *конструкторы класса*. Конструктор экземпляра инициализирует данные экземпляра, конструктор класса — данные класса.

Статический конструктор не имеет параметров, его нельзя вызвать явным образом. Система сама определяет момент, в который требуется его выполнить. Гарантируется только, что это происходит до создания первого экземпляра объекта и до вызова любого статического метода.

Некоторые классы содержат только статические данные, и, следовательно, создавать экземпляры таких объектов не имеет смысла. Чтобы подчеркнуть этот факт, в первой версии C# описывали пустой *закрытый* (private) *конструктор*. Это предотвращало попытки создания экземпляров класса. В листинге 5.7 приведен пример класса, который служит для группировки величин. Создавать экземпляры этого класса запрещено.

Листинг 5.7. Класс со статическим и закрытым конструкторами (для версий ниже 2.0)

```
using System;
namespace ConsoleApplication1
{
    class D
    {
        private D(){}          // закрытый конструктор
        static D()             // статический конструктор
        {
            a = 200;
        }
        static int a;
        static double b = 0.002;
        public static void Print()
        {
            Console.WriteLine( "a = " + a );
            Console.WriteLine( "b = " + b );
        }
        ...
    }
```

продолжение

Листинг 5.7 (*продолжение*)

```
    class Class2
    {   static void Main()
        {
            D.Print();
//          D d = new D();          // ошибка: создать экземпляр невозможно
        }
    }
}
```

ПРИМЕЧАНИЕ

В классе, состоящем только из статических элементов (полей и констант), описывать статический конструктор не обязательно, начальные значения полей удобнее задать при их описании.

В версию 2.0 введена возможность описывать статический класс, то есть класс с модификатором `static`. Экземпляры такого класса создавать запрещено, и кроме того, от него запрещено наследовать. Все элементы такого класса должны явным образом объявляться с модификатором `static` (константы и вложенные типы классифицируются как статические элементы автоматически). Конструктор экземпляра для статического класса задавать, естественно, запрещается.

В листинге 5.8 приведен пример статического класса.

Листинг 5.8. Статический класс (начиная с версии 2.0)

```
using System;
namespace ConsoleApplication1
{
    static class D
    {
        static int a = 200;
        static double b = 0.002;

        public static void Print ()
        {
            Console.WriteLine( "a = " + a );
            Console.WriteLine( "b = " + b );
        }
    }

    class Class1
    {   static void Main()
        {
            D.Print();
        }
    }
}
```

В качестве «сквозного» примера, на котором будет демонстрироваться работа с различными элементами класса, создадим класс, моделирующий персонаж компьютерной игры. Для этого требуется задать его свойства (например, количество щупальцев, силу или наличие гранатомета) и поведение. Естественно, пример (листинг 5.9) является схематичным, поскольку приводится лишь для демонстрации синтаксиса.

Листинг 5.9. Класс Monster

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        public Monster()
        {
            this.name   = "Noname";
            this.health = 100;
            this.ammo   = 100;
        }

        public Monster( string name ) : this()
        {
            this.name = name;
        }

        public Monster( int health, int ammo, string name )
        {
            this.name   = name;
            this.health = health;
            this.ammo   = ammo;
        }

        public string GetName()
        {
            return name;
        }

        public int GetHealth()
        {
            return health;
        }

        public int GetAmmo()
        {
            return ammo;
        }

        public void Passport()
```

продолжение

Листинг 5.9 (*продолжение*)

```
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        string name;                // закрытые поля
        int health, ammo;
    }

    class Class1
    {
        static void Main()
        {
            Monster X = new Monster();
            X.Passport();
            Monster Vasia = new Monster( "Vasia" );
            Vasia.Passport();
            Monster Masha = new Monster( 200, 200, "Masha" );
            Masha.Passport();
        }
    }
}
```

Результат работы программы:

```
Monster Noname   health = 100 ammo = 100
Monster Vasia    health = 100 ammo = 100
Monster Masha    health = 200 ammo = 200
```

В классе три закрытых поля (`name`, `health` и `ammo`), четыре метода (`GetName`, `GetHealth`, `GetAmmo` и `Passport`) и три конструктора, позволяющие задать при создании объекта ни одного, один или три параметра.

Свойства

Свойства служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки. Синтаксис свойства:

```
[ атрибуты ] [ спецификаторы ] тип имя_свойства
{
    [ get код_доступа ]
    [ set код_доступа ]
}
```

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором `public`), поскольку они входят в интерфейс объекта.

Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или установке (set) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно.

Если отсутствует часть set, свойство доступно только для чтения (read-only), если отсутствует часть get, свойство доступно только для записи (write-only).

В версии C# 2.0 введена удобная возможность задавать разные уровни доступа для частей get и set. Например, во многих классах возникает потребность обеспечить неограниченный доступ для чтения и ограниченный — для записи.

Спецификаторы доступа для отдельной части должны задавать либо такой же, либо более ограниченный доступ, чем спецификатор доступа для свойства в целом. Например, если свойство описано как public, его части могут иметь любой спецификатор доступа, а если свойство имеет доступ protected internal, его части могут объявляться как internal, protected или private. Синтаксис свойства в версии 2.0 имеет вид

```
[ атрибуты ] [ спецификаторы ] тип имя_свойства
{
    [ [ атрибуты ] [ спецификаторы ] get код_доступа ]
    [ [ атрибуты ] [ спецификаторы ] set код_доступа ]
}
```

Пример описания свойств:

```
public class Button: Control
{
    private string caption;      // закрытое поле, с которым связано свойство
    public string Caption {      // свойство
       get {                     // способ получения свойства
          return caption;
       }
       set {                       // способ установки свойства
          if (caption != value) {
             caption = value;
          }
       }
    }
    ...
}
```

ПРИМЕЧАНИЕ

Двоеточие между именами Button и Control в заголовке класса Button означает, что класс Button является производным от класса Control.

Метод записи обычно содержит действия по проверке допустимости устанавливаемого значения, метод чтения может содержать, например, поддержку счетчика обращений к полю.

В программе свойство выглядит как поле класса, например:

```
Button ok = new Button();
ok.Caption = "OK";              // вызывается метод установки свойства
string s = ok.Caption;          // вызывается метод получения свойства
```

При обращении к свойству автоматически вызываются указанные в нем методы чтения и установки.

Синтаксически чтение и запись свойства выглядят почти как методы. Метод get должен содержать оператор return, возвращающий выражение, для типа которого должно существовать неявное преобразование к типу свойства. В методе set используется параметр со стандартным именем value, который содержит устанавливаемое значение.

Вообще говоря, свойство может и не связываться с полем. Фактически, оно описывает один или два метода, которые осуществляют некоторые действия над данными того же типа, что и свойство. В отличие от открытых полей, *свойства обеспечивают разделение между внутренним состоянием объекта и его интерфейсом* и, таким образом, упрощают внесение изменений в класс.

С помощью свойств можно отложить инициализацию поля до того момента, когда оно фактически потребуется, например:

```
class A
{
    private static ComplexObject x;      // закрытое поле
    public  static ComplexObject X       // свойство
    {
      get
      {
         if (x == null) {
            x = new ComplexObject(); // создание объекта при 1-м обращении
         }
         return x;
      }
    }
    ...
}
```

Добавим в класс Monster, описанный в листинге 5.9, свойства, позволяющие работать с закрытыми полями этого класса. Свойство Name сделаем доступным только для чтения, поскольку имя объекта задается в конструкторе и его изменение не предусмотрено[1], в свойствах Health и Ammo введем проверку на положительность устанавливаемой величины. Код класса несколько разрастется, зато упростится его использование.

Листинг 5.10. Класс Monster со свойствами

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {   public Monster()
```

[1] Вообще говоря, в данном случае логичнее использовать не свойство, а просто поле со спецификатором readonly. Свойство требуется для демонстрации синтаксиса.

```
{
    this.health = 100;
    this.ammo   = 100;
    this.name   = "Noname";
}

public Monster( string name ) : this()
{
    this.name = name;
}

public Monster( int health, int ammo, string name )
{
    this.health = health;
    this.ammo   = ammo;
    this.name   = name;
}

public int Health          // свойство Health связано с полем health
{
    get
    {
        return health;
    }
    set
    {
        if (value > 0) health = value;
        else           health = 0;
    }
}

public int Ammo                // свойство Ammo связано с полем ammo
{
    get
    {
        return ammo;
    }
    set
    {
        if (value > 0) ammo = value;
        else           ammo = 0;
    }
}

public string Name             // свойство Name связано с полем name
{
    get
    {
        return name;
    }
}
```

продолжение

Листинг 5.10 (*продолжение*)

```
        public void Passport()
        {
            Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
                              name, health, ammo);
        }

        string name;                                    // закрытые поля
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            Monster Masha = new Monster( 200, 200, "Masha" );
            Masha.Passport();
            --Masha.Health;                 // использование свойств
            Masha.Ammo += 100;              // использование свойств
            Masha.Passport();
        }
    }
}
```

Результат работы программы:

```
Monster Masha    health = 200 ammo = 200
Monster Masha    health = 199 ammo = 300
```

Рекомендации по программированию

При создании класса, то есть нового типа данных, следует хорошо продумать его *интерфейс* — средства работы с классом, доступные использующим его программистам. Интерфейс хорошо спроектированного класса интуитивно ясен, непротиворечив и обозрим. Как правило, он не должен включать поля данных.

Поля предпочтительнее делать закрытыми (private). Это дает возможность впоследствии изменить реализацию класса без изменений в его интерфейсе, а также регулировать доступ к полям класса с помощью набора предоставляемых пользователю свойств и методов. Важно помнить, что поля класса вводятся только для того, чтобы реализовать характеристики класса, представленные в его интерфейсе с помощью свойств и методов.

Не нужно расширять интерфейс класса без необходимости, «на всякий случай», поскольку увеличение количества методов затрудняет понимание класса пользователем[1]. В идеале *интерфейс должен быть полным*, то есть предоставлять возможность выполнять любые разумные действия с классом, и одновременно

[1] Под пользователем имеется в виду программист, применяющий класс.

минимально необходимым — без дублирования и пересечения возможностей методов.

Методы определяют поведение класса. *Каждый метод класса должен решать только одну задачу* (не надо объединять два коротких независимых фрагмента кода в один метод). Размер метода может варьироваться в широких пределах, все зависит от того, какие функции он выполняет. Желательно, чтобы тело метода помещалось на 1–2 экрана: одинаково сложно разбираться в программе, содержащей несколько необъятных функций, и в россыпи из сотен единиц по несколько строк каждая.

Если метод реализует сложные действия, следует разбить его на последовательность шагов и каждый шаг оформить в виде вспомогательной функции (метода со спецификатором `private`). Если некоторые действия встречаются в коде хотя бы дважды, их также нужно оформить в виде отдельной функции.

Создание любой функции следует начинать с ее интерфейса, то есть заголовка. Необходимо четко представлять себе, какие параметры функция должна получать и какие результаты формировать. Входные параметры обычно перечисляют в начале списка параметров.

Предпочтительнее, чтобы каждая функция вычисляла ровно один результат, однако это не всегда оправдано. Если величина вычисляется внутри функции и возвращается из нее через список параметров, необходимо использовать перед соответствующим параметром ключевое слово `out`. Если параметр значимого типа может изменить свою величину внутри функции, его предваряют ключевым словом `ref`. Величины ссылочного типа всегда передаются по адресу и, следовательно, могут изменить внутри функции свое значение.

Необходимо стремиться к максимальному сокращению области действия каждой переменной, то есть к реализации принципа *инкапсуляции*. Это упрощает отладку программы, поскольку ограничивает область поиска ошибки. Следовательно, величины, используемые только в функции, следует описывать внутри нее как локальные переменные.

Поля, характеризующие класс в целом, то есть имеющие одно и то же значение для всех экземпляров, следует описывать как статические. Все литералы, связанные с классом (числовые и строковые константы), описываются как поля-константы с именами, отражающими их смысл.

При написании кода методов следует руководствоваться правилами, приведенными в аналогичном разделе главы 4 (см. с. 95).

Глава 6
Массивы и строки

В этой главе рассматриваются элементы, без которых не обходится практически ни одна программа, — массивы и строки.

Массивы

До настоящего момента мы использовали в программах простые переменные. При этом каждой области памяти, выделенной для хранения одной величины, соответствует свое имя. Если переменных много, программа, предназначенная для их обработки, получается длинной и однообразной. Поэтому практически в любом языке есть понятие *массива* — ограниченной совокупности однотипных величин.

Элементы массива имеют одно и то же имя, а различаются порядковым номером (*индексом*). Это позволяет компактно записывать множество операций с помощью циклов.

Массив относится к ссылочным типам данных, то есть располагается в динамической области памяти, поэтому *создание массива* начинается с выделения памяти под его элементы. Элементами массива могут быть величины как значимых, так и ссылочных типов (в том числе массивы). Массив значимых типов хранит значения, массив ссылочных типов — ссылки на элементы. Всем элементам при создании массива присваиваются значения по умолчанию: нули для значимых типов и `null` — для ссылочных.

На рис. 6.1 представлен массив, состоящий из пяти элементов любого значимого типа, например `int` или `double`, а рис. 6.2 иллюстрирует организацию массива из элементов ссылочного типа.

Вот, например, как выглядят операторы создания массива из 10 целых чисел и массива из 100 строк:

```
int[]    w = new int[10];
string[] z = new string[100];
```

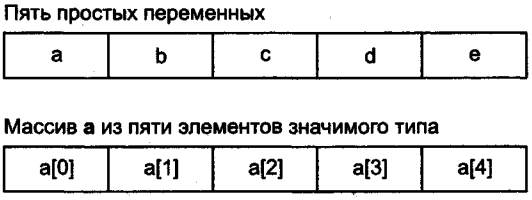


Рис. 6.1. Простые переменные и массив из элементов значимого типа

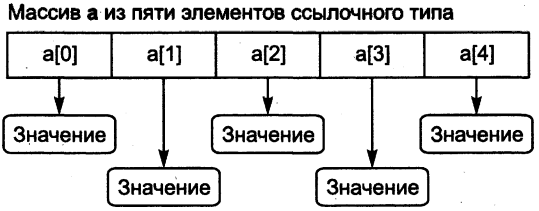


Рис. 6.2. Массив из элементов ссылочного типа

В первом операторе описан массив w типа `int[]`.Операция `new` выделяет память под 10 целых элементов, и они заполняются нулями.

Во втором операторе описан массив z типа `string[]`.Операция `new` выделяет память под 100 ссылок на строки, и эти ссылки заполняются значением `null`. Память под сами строки, составляющие массив, не выделяется — это будет необходимо сделать перед заполнением массива.

Количество элементов в массиве (*размерность*) не является частью его типа, это количество задается при выделении памяти и не может быть изменено впоследствии. Размерность может задаваться не только константой, но и выражением. Результат вычисления этого выражения должен быть неотрицательным, а его тип должен иметь неявное преобразование к `int`, `uint`, `long` или `ulong`.

Пример размерности массива, заданной выражением:

```
short n = ...;
string[] z = new string[n + 1];
```

Элементы массива нумеруются с нуля, поэтому максимальный номер элемента всегда на единицу меньше размерности (например, в описанном выше массиве w элементы имеют индексы от 0 до 9). Для *обращения к элементу массива* после имени массива указывается номер элемента в квадратных скобках, например:

```
w[4]        z[i]
```

С элементом массива можно делать все, что допустимо для переменных того же типа. При работе с массивом автоматически выполняется контроль выхода за его границы: если значение индекса выходит за границы массива, генерируется исключение `IndexOutOfRangeException`.

Массивы одного типа можно присваивать друг другу[1]. При этом происходит присваивание ссылок, а не элементов, как и для любого другого объекта ссылочного типа, например:

```
int[] a = new int[10];
int[] b = a;              // b и a указывают на один и тот же массив
```

Все массивы в C# имеют общий базовый класс `Array`, определенный в пространстве имен `System`. В нем есть несколько полезных методов, упрощающих работу с массивами, например методы получения размерности, сортировки и поиска. Мы рассмотрим эти методы немного позже в разделе «Класс System.Array» (см. с. 133).

ПРИМЕЧАНИЕ

Массивы, являющиеся полями класса, могут иметь те же спецификаторы, что и поля, представляющие собой простые переменные.

В C# существуют три разновидности массивов: одномерные, прямоугольные и ступенчатые (невыровненные).

Одномерные массивы

Одномерные массивы используются в программах чаще всего. Варианты описания массива:

```
тип[] имя;
тип[] имя = new тип [ размерность ];
тип[] имя = { список_инициализаторов };
тип[] имя = new тип [] { список_инициализаторов };
тип[] имя = new тип [ размерность ] { список_инициализаторов };
```

ВНИМАНИЕ

При описании массивов квадратные скобки являются элементом синтаксиса, а не указанием на необязательность конструкции.

Примеры описаний (один пример для каждого варианта описания):

```
int[] a;                                  // 1   элементов нет
int[] b = new int[4];                     // 2   элементы равны 0
int[] c = { 61, 2, 5, -9 };               // 3   new подразумевается
int[] d = new int[] { 61, 2, 5, -9 };     // 4   размерность вычисляется
int[] e = new int[4] { 61, 2, 5, -9 };    // 5   избыточное описание
```

Здесь описано пять массивов. Отличие первого оператора от остальных состоит в том, что в нем, фактически, описана только ссылка на массив, а память под элементы массива не выделена. Если список инициализации не задан, размерность может быть не только константой, но и выражением типа, приводимого к целому.

[1] В более общем виде: можно преобразовать массив, состоящий из элементов некоторого класса `a`, в массив, состоящий из элементов типа, являющегося базовым для `a`.

В каждом из остальных массивов по четыре элемента целого типа. Как видно из операторов 3–5, массив при описании можно *инициализировать*. Если при этом не задана размерность (оператор 4), количество элементов вычисляется по количеству инициализирующих значений. Для полей объектов и локальных переменных можно опускать операцию new, она будет выполнена по умолчанию (оператор 3). Если присутствует и размерность, и список инициализаторов, размерность должна быть константой (оператор 5).

ПРИМЕЧАНИЕ

Если количество инициализирующих значений не совпадает с размерностью, возникает ошибка компиляции.

В качестве примера рассмотрим программу, которая определяет сумму и количество отрицательных элементов, а также максимальный элемент массива, состоящего из 6 целочисленных элементов (листинг 6.1).

Листинг 6.1. Работа с одномерным массивом

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void Main()
        {
            const int n = 6;
            int[] a = new int[n] { 3, 12, 5, -9, 8, -4 };

            Console.WriteLine( "Исходный массив:" );
            for ( int i = 0; i < n; ++i )
                Console.Write( "\t" + a[i] );
            Console.WriteLine();
            long sum = 0;                // сумма отрицательных элементов
            int  num = 0;                // количество отрицательных элементов
            for ( int i = 0; i < n; ++i )
                if ( a[i] < 0 )
                {
                    sum += a[i];
                    ++num;
                }
            Console.WriteLine( "Сумма  отрицательных = " + sum );
            Console.WriteLine( "Кол-во отрицательных = " + num );

            int max = a[0];              // максимальный элемент
            for ( int i = 1; i < n; ++i )
                if ( a[i] > max ) max = a[i];
            Console.WriteLine( "Максимальный элемент = " + max );
        }
    }
}
```

Обратите внимание на то, что для вывода массива требуется организовать цикл.

Прямоугольные массивы

Прямоугольный массив имеет более одного измерения. Чаще всего в программах используются двумерные массивы. Варианты описания двумерного массива:

```
тип[,] имя;
тип[,] имя = new тип [ разм_1, разм_2 ];
тип[,] имя = { список_инициализаторов };
тип[,] имя = new тип [,] { список_инициализаторов };
тип[,] имя = new тип [ разм_1, разм_2 ] { список_инициализаторов };
```

Примеры описаний (один пример для каждого варианта описания):

```
int[,] a;                                    // 1  элементов нет
int[,] b = new int[2, 3];                    // 2  элементы равны 0
int[,] c = {{1, 2, 3}, {4, 5, 6}};           // 3  new подразумевается
int[,] c = new int[,] {{1, 2, 3}, {4, 5, 6}}; // 4  размерность вычисляется
int[,] d = new int[2,3] {{1, 2, 3}, {4, 5, 6}};// 5  избыточное описание
```

Если список инициализации не задан, размерности могут быть не только константами, но и выражениями типа, приводимого к целому. К элементу двумерного массива обращаются, указывая номера строки и столбца, на пересечении которых он расположен, например:

```
a[1, 4]      b[i, j]      b[j, i]
```

ВНИМАНИЕ

Необходимо помнить, что компилятор воспринимает как номер строки первый индекс, как бы он ни был обозначен в программе.

В качестве примера рассмотрим программу, которая для целочисленной матрицы размером 3 × 4 определяет среднее арифметическое ее элементов и количество положительных элементов в каждой строке (рис. 6.3).

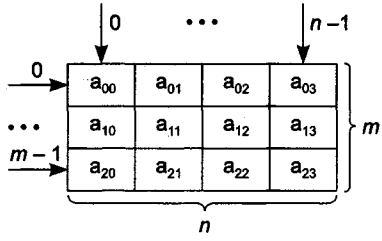


Рис. 6.3. Матрица из *m* строк и *n* столбцов

Для нахождения среднего арифметического элементов массива требуется найти их общую сумму, после чего разделить ее на количество элементов. Порядок перебора элементов массива (по строкам или по столбцам) роли не играет. Нахождение количества положительных элементов каждой строки требует просмотра матрицы по строкам. Схема алгоритма приведена на рис. 6.4, программа — в листинге 6.2.

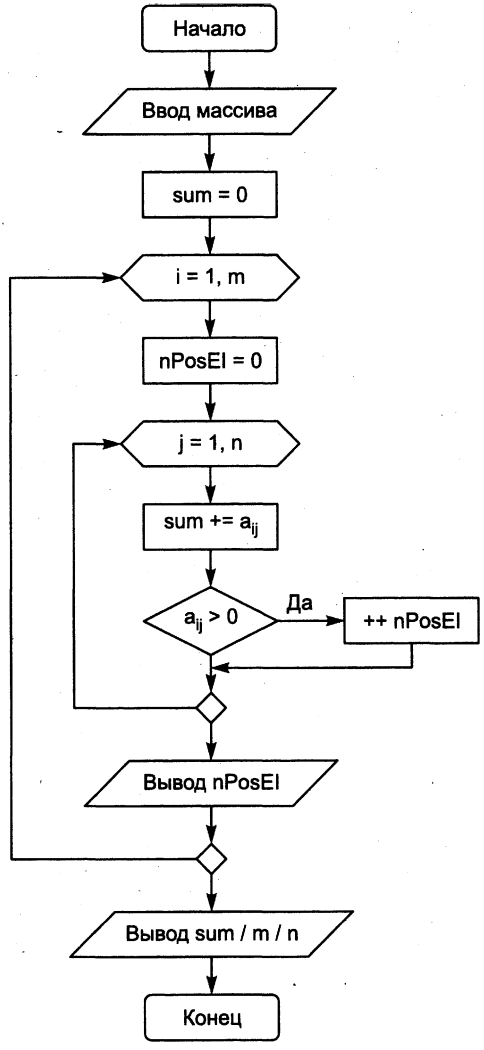


Рис. 6.4. Структурная схема алгоритма для листинга 6.2

Листинг 6.2. Работа с двумерным массивом

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void Main()
        {
            const int m = 3, n = 4;
            int[,] a = new int[m, n] {
                { 2,-2, 8, 9 },
                {-4,-5, 6,-2 },
```

продолжение

Листинг 6.2 (*продолжение*)

```
            { 7, 0, 1, 1 }
        };

        Console.WriteLine( "Исходный массив:" );
        for ( int i = 0; i < m; ++i )
        {
            for ( int j = 0; j < n; ++j )
                Console.Write( "\t" + a[i, j] );
            Console.WriteLine();
        }

        double sum = 0;
        int nPosEl;
        for ( int i = 0; i < m; ++i )
        {
            nPosEl = 0;
            for ( int j = 0; j < n; ++j )
            {
                sum += a[i, j];
                if ( a[i, j] > 0 ) ++nPosEl;
            }
            Console.WriteLine( "В строке {0} {1} положит-х элементов",
                i, nPosEl  );
        }
        Console.WriteLine( "Среднее арифметическое всех элементов: "
            + sum / m / n );
    }
  }
}
```

ПРИМЕЧАНИЕ

Для суммирования элементов описана переменная `sum` вещественного типа. Если описать ее как целую, при делении на количество элементов будет отброшена дробная часть.

Обратите внимание на то, что переменная `sum` обнуляется перед циклом просмотра всей матрицы, а количество положительных элементов — перед циклом просмотра очередной строки, поскольку для каждой строки его вычисление начинается заново.

Ступенчатые массивы

В *ступенчатых массивах* количество элементов в разных строках может различаться. В памяти ступенчатый массив хранится иначе, чем прямоугольный: в виде нескольких внутренних массивов, каждый из которых имеет свой размер. Кроме

того, выделяется отдельная область памяти для хранения ссылок на каждый из внутренних массивов. Организацию ступенчатого массива иллюстрирует рис. 6.5.

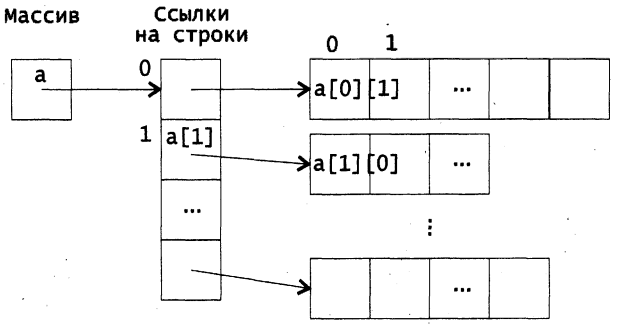


Рис. 6.5. Ступенчатый массив

Описание ступенчатого массива:

```
тип[][] имя;
```

Под каждый из массивов, составляющих ступенчатый массив, память требуется выделять явным образом, например:

```
int[][] a = new int[3][];   // выделение памяти под ссылки на три строки
    a[0] = new int[5];      // выделение памяти под 0-ю строку (5 элементов)
    a[1] = new int[3];      // выделение памяти под 1-ю строку (3 элемента)
    a[2] = new int[4];      // выделение памяти под 2-ю строку (4 элемента)
```

Здесь `a[0]`, `a[1]` и `a[2]` — это отдельные массивы, к которым можно обращаться по имени (пример приведен в следующем разделе). Другой способ выделения памяти:

```
int[][] a = { new int[5], new int[3], new int[4] };
```

К элементу ступенчатого массива обращаются, указывая каждую размерность в своих квадратных скобках, например:

```
a[1][2]        a[i][j]        a[j][i]
```

В остальном использование ступенчатых массивов не отличается от использования прямоугольных. Невыровненные массивы удобно применять, например, для работы с треугольными матрицами большого объема.

Класс System.Array

Ранее уже говорилось, что все массивы в C# построены на основе базового класса `Array`, который содержит полезные для программиста свойства и методы, часть из которых перечислены в табл. 6.1.

Таблица 6.1. Основные элементы класса Array

Элемент	Вид	Описание
Length	Свойство	Количество элементов массива (по всем размерностям)
Rank	Свойство	Количество размерностей массива
BinarySearch	Статический метод	Двоичный поиск в отсортированном массиве
Clear	Статический метод	Присваивание элементам массива значений по умолчанию
Copy	Статический метод	Копирование заданного диапазона элементов одного массива в другой массив
CopyTo	Метод	Копирование всех элементов текущего одномерного массива в другой одномерный массив
GetValue	Метод	Получение значения элемента массива
IndexOf	Статический метод	Поиск первого вхождения элемента в одномерный массив
LastIndexOf	Статический метод	Поиск последнего вхождения элемента в одномерный массив
Reverse	Статический метод	Изменение порядка следования элементов на обратный
SetValue	Метод	Установка значения элемента массива
Sort	Статический метод	Упорядочивание элементов одномерного массива

Свойство Length позволяет реализовывать алгоритмы, которые будут работать с массивами различной длины или, например, со ступенчатым массивом. Использование этого свойства вместо явного задания размерности исключает возможность выхода индекса за границы массива. В листинге 6.3 продемонстрировано применение элементов класса Array при работе с одномерным массивом.

Листинг 6.3. Использование методов класса Array с одномерным массивом

```
using System;
namespace ConsoleApplication1
{    class Class1
    {   static void Main()
        {
            int[] a =  { 24, 50, 18, 3, 16, -7, 9, -1 };

            PrintArray( "Исходный массив:", a );
            Console.WriteLine( Array.IndexOf( a, 18 ) );

            Array.Sort(a);
            PrintArray( "Упорядоченный массив:", a );
            Console.WriteLine( Array.BinarySearch( a, 18) );
        }

        public static void PrintArray( string header, int[] a )
```

```
        {
            Console.WriteLine( header );
            for ( int i = 0; i < a.Length; ++i )
                Console.Write( "\t" + a[i] );
            Console.WriteLine();
        }
    }
}
```

Методы Sort, IndexOf и BinarySearch являются статическими, поэтому к ним обращаются через имя класса, а не экземпляра, и передают в них имя массива. Двоичный поиск можно применять только для упорядоченных массивов. Он выполняется гораздо быстрее, чем линейный поиск, реализованный в методе IndexOf. В листинге поиск элемента, имеющего значение 18, выполняется обоими этими способами.

ПРИМЕЧАНИЕ

Рассмотренные методы имеют по несколько версий (в этом случае употребляется термин «перегруженные методы»), что позволяет выполнять поиск, сортировку и копирование как в массиве целиком, как и в его указанном диапазоне.

В классе Class1 описан вспомогательный статический метод PrintArray, предназначенный для вывода массива на экран. В него передаются два параметра: строка заголовка header и массив. Количество элементов массива определяется внутри метода с помощью свойства Length. Таким образом, этот метод можно использовать для вывода любого целочисленного одномерного массива.

Результат работы программы:

```
Исходный массив:
        24      50      18      3       16      -7      9       -1
2
Упорядоченный массив:
        -7      -1      3       9       16      18      24      50
5
```

Для того чтобы применять метод PrintArray к массивам, состоящим из элементов другого типа, можно описать его второй параметр как Array. Правда, при этом значение элемента массива придется получать с помощью метода GetValue, поскольку доступ по индексу для класса Array не предусмотрен. Обобщенный метод вывода массива выглядит так:

```
public static void PrintArray( string header, Array a )
{
    Console.WriteLine( header );
    for ( int i = 0; i < a.Length; ++i )
        Console.Write( "\t" + a.GetValue(i) );
    Console.WriteLine();
}
```

В листинге 6.4 продемонстрировано применение элементов класса Array при работе со ступенчатым массивом.

Листинг 6.4. Использование методов класса Array со ступенчатым массивом

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            int[][] a = new int[3][];
            a[0] = new int [5] { 24, 50, 18, 3, 16 };
            a[1] = new int [3] { 7, 9, -1 };
            a[2] = new int [4] { 6, 15, 3, 1 };

            Console.WriteLine( "Исходный массив:" );
            for ( int i = 0; i < a.Length; ++i )
            {
                for ( int j = 0; j < a[i].Length; ++j )
                    Console.Write( "\t" + a[i][j] );
                Console.WriteLine();
            }
            Console.WriteLine( Array.IndexOf( a[0], 18 ) );
        }
    }
}
```

Обратите внимание на то, как внутри цикла по строкам определяется длина каждого массива. Результат работы программы:

```
Исходный массив:
        24      50      18      3       16
        7       9       -1
        6       15      3       1
2
```

Оператор foreach

Оператор foreach применяется для перебора элементов в специальным образом организованной группе данных. Массив является именно такой группой[1]. Удобство этого вида цикла заключается в том, что нам не требуется определять количество элементов в группе и выполнять их перебор по индексу: мы просто указываем на необходимость перебрать все элементы группы. Синтаксис оператора:

```
foreach ( тип имя in выражение ) тело_цикла
```

[1] Отчего это так, вы поймете, изучив главу 9.

Имя задает локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из массива *выражение* (в качестве выражения чаще всего применяется имя массива или другой группы данных). В простом или составном операторе, представляющем собой *тело цикла*, выполняются действия с переменной цикла. Тип переменной должен соответствовать типу элемента массива.

Например, пусть задан массив:

```
int[] a =  { 24, 50, 18, 3, 16, -7, 9, -1 };
```

Вывод этого массива на экран с помощью оператора foreach выглядит следующим образом:

```
foreach ( int x in a ) Console.WriteLine( x );
```

Этот оператор выполняется так: на каждом проходе цикла очередной элемент массива присваивается переменной x и с ней производятся действия, записанные в теле цикла.

Ступенчатый массив из листинга 6.4 вывести на экран с помощью оператора foreach немного сложнее, чем одномерный, но все же проще, чем с помощью цикла for:

```
foreach ( int[] x in a )
{
    foreach ( int y in x ) Console.Write( "\t" + y );
    Console.WriteLine();
}
```

В листинге 6.5 решается та же задача, что и в листинге 6.1, но с использованием цикла foreach. Обратите внимание на то, насколько понятнее стала программа.

Листинг 6.5. Работа с одномерным массивом с использованием цикла foreach

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            int[] a = { 3, 12, 5, -9, 8, -4 };

            Console.WriteLine( "Исходный массив:" );
            foreach ( int elem in a )
                Console.Write( "\t" + elem );
            Console.WriteLine();

            long sum = 0;              // сумма отрицательных элементов
            int  num = 0;              // количество отрицательных элементов
            foreach ( int elem in a )
                if ( elem < 0 )
                {
```

продолжение

Листинг 6.5 (*продолжение*)

```
                sum += elem;
                ++num;
            }

        Console.WriteLine( "sum = " + sum );
        Console.WriteLine( "num = " + num );

        int max = a[0];            // максимальный элемент
        foreach ( int elem in a )
            if ( elem > max ) max = elem;

        Console.WriteLine( "max = " + max );
    }
  }
}
```

А вот как можно переписать метод вывода массива из листинга 6.3:

```
public static void PrintArray( string header, Array a )
{
    Console.WriteLine( header );
    foreach ( object x in a ) Console.Write( "\t" + x );
    Console.WriteLine();
}
```

Такая запись становится возможной потому, что любой объект может быть неявно преобразован к типу его базового класса, а тип `object`, как вы помните, является корневым классом всей иерархии. Когда вы продвинетесь в изучении C# до раздела «Виртуальные методы» (см. с. 178), вы поймете механизм выполнения этого кода, а пока можете просто им пользоваться.

ВНИМАНИЕ

Ограничением оператора `foreach` является то, что с его помощью можно только просматривать значения в группе данных, но не изменять их.

Массивы объектов

При создании массива, состоящего из элементов ссылочного типа, память выделяется только под ссылки на элементы, а сами элементы необходимо разместить в хипе явным образом. В качестве примера создадим массив из объектов некоторого класса `Monster`:

```
using System;
namespace ConsoleApplication1
{
    class Monster { ... }

    class Class1
```

```
    {   static void Main()
        {
            Random rnd = new Random();
            const int n = 5;
            Monster[] stado = new Monster[n];                    // 1
            for ( int i = 0; i < n; ++i )                        // 2
            {
                stado[i] = new Monster( rnd.Next( 1, 100 ),
                                        rnd.Next( 1, 200 ),
                                       "Crazy" + i.ToString() );
            }
            foreach ( Monster x in stado ) x.Passport();         // 3
        }
    }
}
```

Результат работы программы:

```
Monster Crazy0   health = 18 ammo = 94
Monster Crazy1   health = 85 ammo = 75
Monster Crazy2   health = 13 ammo = 6
Monster Crazy3   health = 51 ammo = 104
Monster Crazy4   health = 68 ammo = 114
```

В программе для получения случайных значений использован стандартный класс Random, который описан далее в этой главе (см. с. 148). В операторе 1 выделяется пять ячеек памяти под ссылки на экземпляры класса Monster (из листинга 5.9), и эти ссылки заполняются значением null. В цикле 2 создаются пять объектов: операция new выделяет память в хипе необходимого для хранения полей объекта объема, а конструктор объекта заносит в эти поля соответствующие значения (выполняется версия конструктора с тремя параметрами). Цикл 3 демонстрирует удобство применения оператора foreach для работы с массивом.

Символы и строки

Обработка текстовой информации является, вероятно, одной из самых распространенных задач в современном программировании, и C# предоставляет для ее решения широкий набор средств: отдельные символы, массивы символов, изменяемые и неизменяемые строки и регулярные выражения.

Символы

Символьный тип char предназначен для хранения символов в кодировке Unicode. Способы представления символов рассматривались в разделе «Литералы» (см. с. 26). Символьный тип относится к встроенным типам данных C# и соответствует стандартному классу Char библиотеки .NET из пространства имен System. В этом классе определены статические методы, позволяющие задать вид и категорию символа,

а также преобразовать символ в верхний или нижний регистр и в число. Основные методы приведены в табл. 6.2.

Таблица 6.2. Основные методы класса System.Char

Метод	Описание
GetNumericValue	Возвращает числовое значение символа, если он является цифрой, и –1 в противном случае
GetUnicodeCategory	Возвращает категорию Unicode-символа[1]
IsControl	Возвращает true, если символ является управляющим
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой
IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsNumber	Возвращает true, если символ является числом (десятичным или шестнадцатеричным)
IsPunctuation	Возвращает true, если символ является знаком препинания
IsSeparator	Возвращает true, если символ является разделителем
IsUpper	Возвращает true, если символ записан в верхнем регистре
IsWhiteSpace	Возвращает true, если символ является пробельным (пробел, перевод строки и возврат каретки)
Parse	Преобразует строку в символ (строка должна состоять из одного символа)
ToLower	Преобразует символ в нижний регистр
ToUpper	Преобразует символ в верхний регистр
MaxValue, MinValue	Возвращают символы с максимальным и минимальным кодами (эти символы не имеют видимого представления)

В листинге 6.6 продемонстрировано использование этих методов.

Листинг 6.6. Использование методов класса System.Char

```
using System;
namespace ConsoleApplication1
{  class Class1
   {  static void Main()
      {
         try
         {
            char b = 'B', c = '\x63', d = '\u0032';                    // 1
```

[1] Все Unicode-символы разделены на категории, например, десятичные цифры (DecimalDigitNumber), римские цифры (LetterNumber), разделители строк (LineSeparator), буквы в нижнем регистре (LowercaseLetter) и т. д.

```
            Console.WriteLine( "{0}  {1}  {2}", b, c, d );
            Console.WriteLine( "{0}  {1}  {2}",
                 char.ToLower(b), char.ToUpper(c), char.GetNumericValue(d) );

            char a;
            do                                                          // 2
            {
                Console.Write( "Введите символ: " );
                a = char.Parse( Console.ReadLine() );
                Console.WriteLine( "Введен символ {0}, его код - {1}",
                                   a, (int)a );
                if (char.IsLetter(a))      Console.WriteLine("Буква");
                if (char.IsUpper(a))       Console.WriteLine("Верхний рег.");
                if (char.IsLower(a))       Console.WriteLine("Нижний рег.");
                if (char.IsControl(a))     Console.WriteLine("Управляющий");
                if (char.IsNumber(a))      Console.WriteLine("Число");
                if (char.IsPunctuation(a)) Console.WriteLine("Разделитель");
            } while (a != 'q');
        }
        catch
        {
            Console.WriteLine( "Возникло исключение" );
            return;
        }
    }
  }
}
```

В операторе 1 описаны три символьных переменных. Они инициализируются символьными литералами в различных формах представления. Далее выполняются вывод и преобразование символов.

В цикле 2 анализируется вводимый с клавиатуры символ. Можно вводить и управляющие символы, используя сочетание клавиши Ctrl с латинскими буквами. При вводе использован метод Parse, преобразующий строку, которая должна содержать единственный символ, в символ типа char. Поскольку вводится строка, ввод каждого символа следует завершать нажатием клавиши Enter. Цикл выполняется, пока пользователь не введет символ q.

Вывод символа сопровождается его кодом в десятичном виде[1]. Для вывода кода используется явное преобразование к целому типу. Явное преобразование из символов в строки и обратно в C# не существует, неявным же образом любой объект, в том числе и символ, может быть преобразован в строку[2], например:

```
string s = 'к' + 'о' + 'т';              // результат - строка "кот"
```

При вводе и преобразовании могут возникать исключительные ситуации, например, если пользователь введет пустую строку. Для «мягкого» завершения программы предусмотрена обработка исключений.

---

[1] Обратите внимание на коды русских букв: как далеко от латинских они оказались в кодировке Unicode!

[2] Об этом рассказывалось в разделе «Простейший ввод-вывод» (см. с. 59).

Массивы символов

Массив символов, как и массив любого иного типа, построен на основе базового класса Array, некоторые свойства и методы которого были перечислены в табл. 6.1. Применение этих методов позволяет эффективно решать некоторые задачи. Простой пример приведен в листинге 6.7.

Листинг 6.7. Работа с массивом символов

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            char[] a = { 'm', 'a', 's', 's', 'i', 'v' };                    // 1
            char[] b = "а роза упала на лапу азора".ToCharArray();          // 2

            PrintArray( "Исходный массив a:", a );

            int pos = Array.IndexOf( a, 'm' );
            a[pos] = 'M';
            PrintArray( "Измененный массив a:", a );

            PrintArray( "Исходный массив b:", b );
            Array.Reverse( b );
            PrintArray( "Измененный массив b:", b );
        }

        public static void PrintArray( string header, Array a )
        {
            Console.WriteLine( header );
            foreach ( object x in a ) Console.Write( x );
            Console.WriteLine( "\n" );
        }
    }
}
```

Результат работы программы:

```
Исходный массив a:
massiv

Измененный массив a:
Massiv

Исходный массив b:
а роза упала на лапу азора

Измененный массив b:
ароза упал ан алапу азор а
```

Символьный массив можно инициализировать, либо непосредственно задавая его элементы (оператор 1), либо применяя метод ToCharArray класса string, который разбивает исходную строку на отдельные символы (оператор 2).

Строки типа string

Тип string, предназначенный для работы со строками символов в кодировке Unicode, является встроенным типом C#. Ему соответствует базовый класс System.String библиотеки .NET.

Создать строку можно несколькими способами:

```
string s;                       // инициализация отложена
string t = "qqq";               // инициализация строковым литералом
string u = new string(' ', 20); // конструктор создает строку из 20 пробелов
char[] a = { '0', '0', '0' };   // массив для инициализации строки
string v = new string( a );     // создание из массива символов
```

Для строк определены следующие *операции*:

- присваивание (=);
- проверка на равенство (==);
- проверка на неравенство (!=);
- обращение по индексу ([]);
- сцепление (конкатенация) строк (+).

Несмотря на то что строки являются ссылочным типом данных, на равенство и неравенство проверяются не ссылки, а значения строк. Строки равны, если имеют одинаковое количество символов и совпадают посимвольно.

Обращаться к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения. Это связано с тем, что строки типа string относятся к так называемым *неизменяемым типам данных*[1]. Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые «старые» копии автоматически удаляются сборщиком мусора.

В классе System.String предусмотрено множество методов, полей и свойств, позволяющих выполнять со строками практически любые действия. Основные элементы класса приведены в табл. 6.3.

Таблица 6.3. Основные элементы класса System.String

Название	Вид	Описание
Compare	Статический метод	Сравнение двух строк в лексикографическом (алфавитном) порядке. Разные реализации метода позволяют сравнивать строки и подстроки с учетом и без учета регистра и особенностей национального представления дат и т. д.

продолжение

[1] Наследовать от этого класса также запрещается.

Таблица 6.3 (*продолжение*)

Название	Вид	Описание
`CompareOrdinal`	Статический метод	Сравнение двух строк по кодам символов. Разные реализации метода позволяют сравнивать строки и подстроки
`CompareTo`	Метод	Сравнение текущего экземпляра строки с другой строкой
`Concat`	Статический метод	Конкатенация строк. Метод допускает сцепление произвольного числа строк
`Copy`	Статический метод	Создание копии строки
`Empty`	Статическое поле	Пустая строка (только для чтения)
`Format`	Статический метод	Форматирование в соответствии с заданными спецификаторами формата (см. далее)
`IndexOf`, `IndexOfAny`, `LastIndexOf`, `LastIndexOfAny`	Методы	Определение индексов первого и последнего вхождения заданной подстроки или любого символа из заданного набора
`Insert`	Метод	Вставка подстроки в заданную позицию
`Intern`, `IsInterned`	Статические методы	Возвращает ссылку на строку, если такая уже существует. Если строки нет, `Intern` добавляет строку во внутренний пул, `IsInterned` возвращает `null`
`Join`	Статический метод	Слияние массива строк в единую строку. Между элементами массива вставляются разделители (см. далее)
`Length`	Свойство	Длина строки (количество символов)
`PadLeft`, `PadRight`	Методы	Выравнивание строки по левому или правому краю путем вставки нужного числа пробелов в начале или в конце строки
`Remove`	Метод	Удаление подстроки из заданной позиции
`Replace`	Метод	Замена всех вхождений заданной подстроки или символа новыми подстрокой или символом
`Split`	Метод	Разделяет строку на элементы, используя заданные разделители. Результаты помещаются в массив строк
`StartsWith`, `EndsWith`	Методы	Возвращает `true` или `false` в зависимости от того, начинается или заканчивается строка заданной подстрокой
`Substring`	Метод	Выделение подстроки, начиная с заданной позиции
`ToCharArray`	Метод	Преобразование строки в массив символов
`ToLower`, `ToUpper`	Методы	Преобразование символов строки к нижнему или верхнему регистру

Название	Вид	Описание
Trim, TrimStart, TrimEnd	Методы	Удаление пробелов в начале и конце строки или только с одного ее конца (обратные по отношению к методам PadLeft и PadRight действия)

Пример применения методов приведен в листинге 6.8.

Листинг 6.8. Работа со строками типа string

```
using System;
namespace ConsoleApplication1,
{   class Class1
    {   static void Main()
        {
            string s = "прекрасная королева Изольда";
            Console.WriteLine( s );
            string sub = s.Substring( 3 ).Remove( 12, 2 );              // 1
            Console.WriteLine( sub );

            string[] mas = s.Split(' ');                                // 2
            string joined = string.Join( "! ", mas );
            Console.WriteLine( joined );

            Console.WriteLine( "Введите строку" );
            string x = Console.ReadLine();                              // 3
            Console.WriteLine( "Вы ввели строку " + x );

            double a = 12.234;
            int b = 29;
            Console.WriteLine( " a = {0,6:C}  b = {1,2:X}", a, b );     // 4
            Console.WriteLine( " a = {0,6:0.##}  b = {1,5:0.# ' руб. '}",
                               a, b );                                  // 5
            Console.WriteLine(" a = {0:F3} b = {1:D3}", a, b,);         // 6
        }
    }
}
```

Результат работы программы:

```
прекрасная королева Изольда
красная корова Изольда
прекрасная! королева! Изольда
Введите строку
не хочу!
Вы ввели строку не хочу!
a = 12,23р.  b = 1D
a = 12,23    b = 29 руб.
```

В операторе 1 выполняются два последовательных вызова методов: метод Substring возвращает подстроку строки s, которая содержит символы исходной строки, начиная с третьего. Для этой подстроки вызывается метод Remove, удаляющий из нее два символа, начиная с 12-го. Результат работы метода присваивается переменной sub.

Аргументом метода Split (оператор 2) является разделитель, в данном случае — символ пробела. Метод разделяет строку на отдельные слова, которые заносятся в массив строк mas. Статический метод Join (он вызывается через имя класса) объединяет элементы массива mas в одну строку, вставляя между каждой парой слов строку "! ". Оператор 3 напоминает вам о том, как вводить строки с клавиатуры. В операторах 4-6 используется форматирование строк, более подробно описанное в следующем разделе.

Форматирование строк

В операторе 4 из листинга 6.8 неявно применяется метод Format, который заменяет все вхождения параметров в фигурных скобках значениями соответствующих переменных из списка вывода. После номера параметра можно задать минимальную ширину поля вывода, а также указать спецификатор формата, который определяет форму представления выводимого значения.

В общем виде параметр задается следующим образом:

```
{n [,m[:спецификатор_формата[число]]]}
```

Здесь n — номер параметра. Параметры нумеруются с нуля, нулевой параметр заменяется значением первой переменной из списка вывода, первый параметр — второй переменной и т. д. Параметр m определяет минимальную ширину поля, которое отводится под выводимое значение. Если выводимому числу достаточно меньшего количества позиций, неиспользуемые позиции заполняются пробелами. Если числу требуется больше позиций, параметр игнорируется.

Спецификатор формата, как явствует из его названия, определяет формат вывода значения. Например, спецификатор C (Currency) означает, что параметр должен форматироваться как валюта с учетом национальных особенностей представления, а спецификатор X (Hexadecimal) задает шестнадцатеричную форму представления выводимого значения. После некоторых спецификаторов можно задать количество позиций, отводимых под дробную часть выводимого значения (см. оператор 6).

ПРИМЕЧАНИЕ

До настоящего момента мы пользовались сокращенной записью, задавая только номера параметров. Список спецификаторов формата приведен в приложении.

В операторе 5 из листинга 6.8 используются *пользовательские шаблоны форматирования*. Если приглядеться, в них нет ничего сложного: после двоеточия задается вид выводимого значения посимвольно, причем на месте каждого символа может стоять либо #, либо 0. Если указан знак #, на этом месте будет выведена цифра числа, если она не равна нулю. Если указан 0, будет выведена любая цифра, в том числе и незначащий 0. В табл. 6.4 приведены примеры шаблонов и результатов вывода.

Таблица 6.4. Примеры применения пользовательских шаблонов форматирования

Число	Шаблон	Вид
1,243	00.00	01,24
1,243	#.##	1,24
0,1	00.00	00,10
0,1	#.##	,1

Пользовательский шаблон может также содержать текст, который в общем случае заключается в апострофы.

Строки типа StringBuilder

Возможности, предоставляемые классом string, широки, однако требование неизменности его объектов может оказаться неудобным. В этом случае для работы со строками применяется класс StringBuilder, определенный в пространстве имен System.Text и позволяющий изменять значение своих экземпляров.

При создании экземпляра обязательно использовать операцию new и конструктор, например:

```
StringBuilder a = new StringBuilder();                          // 1
StringBuilder b = new StringBuilder( "qwerty" );                // 2
StringBuilder c = new StringBuilder( 100 );                     // 3
StringBuilder d = new StringBuilder( "qwerty", 100 );           // 4
StringBuilder e = new StringBuilder( "qwerty", 1, 3, 100 );     // 5
```

В конструкторе класса указываются два вида параметров: инициализирующая строка или подстрока и объем памяти, отводимой под экземпляр (*емкость буфера*). Один или оба параметра могут отсутствовать, в этом случае используются их значения по умолчанию.

Если применяется конструктор без параметров (оператор 1), создается пустая строка размера, заданного по умолчанию (16 байт). Другие виды конструкторов задают объем памяти, выделяемой строке, и/или ее начальное значение. Например, в операторе 5 объект инициализируется подстрокой длиной 3 символа, начиная с первого (подстрока "wer"). Основные элементы класса StringBuilder приведены в табл. 6.5.

Таблица 6.5. Основные элементы класса System.Text.StringBuilder

Название	Вид	Описание
Append	Метод	Добавление в конец строки. Разные варианты метода позволяют добавлять в строку величины любых встроенных типов, массивы символов, строки и подстроки типа string
AppendFormat	Метод	Добавление форматированной строки в конец строки
Capacity	Свойство	Получение или установка емкости буфера. Если устанавливаемое значение меньше текущей длины строки или больше максимального, генерируется исключение ArgumentOutOfRangeException
Insert	Метод	Вставка подстроки в заданную позицию
Length	Свойство	Длина строки (количество символов)
MaxCapacity	Свойство	Максимальный размер буфера
Remove	Метод	Удаление подстроки из заданной позиции

продолжение

Таблица 6.5 (*продолжение*)

Название	Вид	Описание
Replace	Метод	Замена всех вхождений заданной подстроки или символа новой подстрокой или символом
ToString	Метод	Преобразование в строку типа string

Пример применения методов приведен в листинге 6.9.

Листинг 6.9. Работа со строками типа StringBuilder

```
using System;
using System.Text;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            Console.Write( "Введите зарплату: " );
            double salary = double.Parse( Console.ReadLine() );

            StringBuilder a = new StringBuilder();
            a.Append( "зарплата " );
            a.AppendFormat( "{0, 6:C} - в год {1, 6:C}",
                            salary, salary * 12 );
            Console.WriteLine( a );

            a.Replace( "р.", "тыс.$" );
            Console.WriteLine( "А лучше было бы: " + a );
        }
    }
}
```

Результат работы программы:

```
Введите зарплату: 3500
зарплата 3 500,00р. - в год 42 000,00р.
А лучше было бы: зарплата 3 500,00тыс.$ - в год 42 000,00тыс.$
```

Емкость буфера не соответствует количеству символов в строке и может увеличиваться в процессе работы программы как в результате прямых указаний программиста, так и вследствие выполнения методов изменения строки, если строка в результате превышает текущий размер буфера. Программист может уменьшить размер буфера с помощью свойства Capacity, чтобы не занимать лишнюю память.

Класс Random

При отладке программ, использующих массивы, удобно иметь возможность генерировать исходные данные, заданные случайным образом. В библиотеке C# на этот случай есть класс Random, определенный в пространстве имен System.

Для получения псевдослучайной последовательности чисел необходимо сначала создать экземпляр класса с помощью конструктора, например:

```
Random a = new Random();        // 1
Random b = new Random( 1 );     // 2
```

Есть два вида конструктора: конструктор без параметров (оператор 1) использует начальное значение генератора, вычисленное на основе текущего времени. В этом случае каждый раз создается уникальная последовательность. Конструктор с параметром типа int (оператор 2) задает начальное значение генератора, что обеспечивает возможность получения одинаковых последовательностей чисел.

Для получения очередного значения серии пользуются методами, перечисленными в табл. 6.6.

Таблица 6.6. Основные методы класса System.Random

Название	Описание
Next()	Возвращает целое положительное число во всем положительном диапазоне типа int
Next(макс)	Возвращает целое положительное число в диапазоне [0, макс]
Next(мин, макс)	Возвращает целое положительное число в диапазоне [мин, макс]
NextBytes(массив)	Возвращает массив чисел в диапазоне [0, 255]
NextDouble()	Возвращает вещественное положительное число в диапазоне [0, 1)

Пример применения методов приведен в листинге 6.10.

Листинг 6.10. Работа с генератором псевдослучайных чисел

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            Random a = new Random();
            Random b = new Random( 1 );
            const int n = 10;

            Console.WriteLine( "\n  Диапазон [0, 1]:" );
            for ( int i = 0; i < n; ++i )
                Console.Write( "{0 ,6:0.##}", a.NextDouble() );

            Console.WriteLine( "\n  Диапазон [0, 1000]:" );
            for ( int i = 0; i < n; ++i )
                Console.Write( "    " + b.Next( 1000 ) );

            Console.WriteLine( "\n  Диапазон [-10, 10]:" );
```

продолжение

Листинг 6.10 (*продолжение*)

```
            for ( int i = 0; i < n; ++i )
                Console.Write( "   " + a.Next(-10, 10) );

            Console.WriteLine( "\n  Массив [0, 255]:" );
            byte[] mas = new byte[n];
            a.NextBytes( mas );
            for (int i = 0; i < n; ++i) Console.Write( " " + mas[i] );
        }
    }
}
```

Результат работы программы:

```
Диапазон [0, 1]:
0,02   0,4  0,24  0,55  0,92  0,84   0,9  0,78  0,78  0,74
Диапазон [0, 1000]:
 248   110   467   771   657   432   354   943   101   642
Диапазон [-10, 10]:
 -8   9   -6   -10   7   4   9   -5   -2   -1
Массив [0, 255]:
181 105 60 50 70 77 9 28 133 150
```

Рекомендации по программированию

Используйте для хранения данных массив, если количество однотипных элементов, которые требуется обработать в вашей программе, известно или, по крайней мере, известно максимальное количество таких элементов. В последнем случае память под массив выделяется «по максимуму», а фактическое количество элементов хранится в отдельной переменной, которая вычисляется в программе.

При работе с массивом нужно обязательно предусматривать обработку исключения `IndexOutOfRangeException`, если индекс для обращения к массиву вычисляется в программе по формулам, а не задается с помощью констант или счетчиков циклов `for`.

СОВЕТ

При отладке программ, использующих массивы, исходные данные удобно подготовить в текстовом файле. Это позволит продумать, какие значения элементов необходимо задать, чтобы протестировать выполнение каждой ветви программы. Например, исходные данные для программы, вычисляющей в одномерном массиве номер первого элемента, равного нулю, должны включать варианты, когда такой элемент встречается в массиве ни одного, один и более раз.

Если количество элементов, обрабатываемых программой, может быть произвольным, удобнее использовать не массив, а другие структуры данных, например параметризованные коллекции, которые рассматриваются в главе 13.

При работе со строками необходимо учитывать, что в C# строка типа `string` является неизменяемым типом данных, то есть любая операция изменения строки на самом деле возвращает ее копию. Для изменения строк используется тип `StringBuilder`. Прежде чем описывать в программе какое-либо действие со строками, полезно посмотреть, нет ли в списке элементов используемого класса подходящих методов и свойств.

Для эффективного поиска и преобразования текста в соответствии с заданными шаблонами используются так называемые *регулярные выражения*, которые рассмотрены в главе 15.

Глава 7
Классы: подробности

В этой главе мы продолжим знакомство с элементами классов, начатое в главе 5. Сначала мы рассмотрим дополнительные возможности методов, не описанные в главе 5, а затем перейдем к новым элементам класса — индексаторам, операциям и деструкторам.

Перегрузка методов

Часто бывает удобно, чтобы методы, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Если имя метода является осмысленным и несет нужную информацию, это делает программу более понятной, поскольку для каждого действия требуется помнить только одно имя. Использование нескольких методов с одним и тем же именем, но различными типами параметров называется *перегрузкой методов*.

ПРИМЕЧАНИЕ

Мы уже использовали перегруженные версии методов стандартных классов и даже сами создавали перегруженные методы, когда описывали в классе несколько конструкторов.

Компилятор определяет, какой именно метод требуется вызвать, по типу фактических параметров. Этот процесс называется *разрешением* (resolution) перегрузки. Тип возвращаемого методом значения в разрешении не участвует[1]. Механизм разрешения основан на достаточно сложном наборе правил, смысл которых сводится к тому, чтобы использовать метод с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта метода, определяющего наибольшее значение:

[1] Если последний параметр метода имеет модификатор `params`, о котором рассказывается далее в этой главе, этот параметр также не учитывается.

```
// Возвращает наибольшее из двух целых:
int max( int a, int b )
// Возвращает наибольшее из трех целых:
int max( int a, int b, int c )
// Возвращает наибольшее из первого параметра и длины второго:
int max ( int a, string b )
// Возвращает наибольшее из второго параметра и длины первого:
int max ( string b, int a )
...
Console.WriteLine( max( 1, 2 ) );
Console.WriteLine( max( 1, 2, 3 ) );
Console.WriteLine( max( 1, "2" ) );
Console.WriteLine( max( "1", 2 ) );
```

При вызове метода `max` компилятор выбирает вариант метода, соответствующий типу передаваемых в метод аргументов (в приведенном примере будут последовательно вызваны все четыре варианта метода).

Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами, например, `bool` и `char` в `int`, `float` в `double` и т. п. Если преобразование невозможно, выдается сообщение об ошибке. Если соответствие на одном и том же этапе может быть получено более чем одним способом, выбирается «лучший» из вариантов, то есть вариант, содержащий меньшие количество и длину преобразований в соответствии с правилами, описанными в разделе «Преобразования встроенных арифметических типов-значений» (см. с. 45). Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

Вам уже известно, что все методы класса должны различаться сигнатурами. Это понятие было введено в разделе «Методы» (см. с. 106). Перегруженные методы имеют одно имя, но должны различаться параметрами, точнее, их типами и способами передачи (`out` или `ref`). Например, методы, заголовки которых приведены ниже, имеют различные сигнатуры и считаются перегруженными:

```
int max( int a, int b )
int max( int a, ref int b )
```

Перегрузка методов является проявлением *полиморфизма*, одного из основных свойств ООП. Программисту гораздо удобнее помнить одно имя метода и использовать его для работы с различными типами данных, а решение о том, какой вариант метода вызвать, возложить на компилятор. Этот принцип широко используется в классах библиотеки .NET. Например, в стандартном классе `Console` метод `WriteLine` перегружен 19 раз для вывода величин разных типов.

Рекурсивные методы

Рекурсивным называется метод, который вызывает сам себя. Такая рекурсия называется *прямой*. Существует еще *косвенная* рекурсия, когда два или более метода вызывают друг друга. Если метод вызывает себя, в стеке создается копия значений

его параметров, как и при вызове обычного метода, после чего управление передается первому исполняемому оператору метода. При повторном вызове этот процесс повторяется.

Ясно, что для завершения вычислений каждый рекурсивный метод должен содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении метода соответствующая часть стека освобождается и управление передается вызывающему методу, выполнение которого продолжается с точки, следующей за рекурсивным вызовом.

Классическим примером рекурсивной функции является функция вычисления факториала (это не означает, что факториал следует вычислять именно так). Для того чтобы получить значение факториала числа n, требуется умножить на n факториал числа $(n - 1)$. Известно также, что $0! = 1$ и $1! = 1$:

```
long fact( long n ) {
    if ( n == 0 || n == 1 ) return 1;          // нерекурсивная ветвь
    return ( n * fact( n - 1 ) );              // рекурсивная ветвь
}
```

То же самое можно записать короче:

```
long fact( long n ) {
    return ( n > 1 ) ? n * fact( n - 1 ) : 1;
}
```

Рекурсивные методы чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно, например, с двоичными деревьями (понятие о двоичном дереве дается в главе 13). Любой рекурсивный метод можно реализовать без применения рекурсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно.

К *достоинствам* рекурсии можно отнести компактность записи, к *недостаткам* — расход времени и памяти на повторные вызовы метода и передачу ему копий параметров, а главное, опасность переполнения стека.

Методы с переменным количеством аргументов

Иногда бывает удобно создать метод, в который можно передавать разное количество аргументов. Язык C# предоставляет такую возможность с помощью ключевого слова `params`. Параметр, помеченный этим ключевым словом, размещается в списке параметров последним и обозначает массив заданного типа неопределенной длины, например:

```
public int Calculate( int a, out int c, params int[] d ) …
```

В этот метод можно передать три и более параметров. Внутри метода к параметрам, начиная с третьего, обращаются как к обычным элементам массива. Количество

элементов массива получают с помощью его свойства Length. В качестве примера рассмотрим метод вычисления среднего значения элементов массива (листинг 7.1).

Листинг 7.1. Метод с переменным числом параметров

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        public static double Average( params int[] a )
        {
            if ( a.Length == 0 )
                throw new Exception( "Недостаточно аргументов в методе" );

            double av = 0;
            foreach ( int elem in a ) av += elem;
            return av / a.Length;
        }

        static void Main()
        {   try
            {
                int[] a = { 10, 20, 30 };
                Console.WriteLine( Average( a ) );              // 1
                int[] b = { -11, -4, 12, 14, 32, -1, 28 };
                Console.WriteLine( Average( b ) );              // 2
                short z = 1, e = 13;
                byte v = 107;
                Console.WriteLine( Average( z, e, v ) );        // 3
                Console.WriteLine( Average() );                 // 4
            }
            catch( Exception e )
            {
                Console.WriteLine( e.Message );
                return;
            }
        }
    }
}
```

Результат работы программы:

```
20
10
38
Недостаточно аргументов в методе
```

ПРИМЕЧАНИЕ

В данном алгоритме отсутствие аргументов при вызове метода Average является ошибкой. Этот случай обрабатывается генерацией исключения. Если не обработать эту ошибку, результат вычисления среднего будет равен «не числу» (NaN) вследствие деления на ноль в операторе возврата из метода.

Параметр-массив может быть только один и должен располагаться последним в списке. Соответствующие ему аргументы должны иметь типы, для которых возможно неявное преобразование к типу массива.

Метод Main

Метод, которому передается управление после запуска программы, должен иметь имя Main и быть статическим. Он может *принимать параметры* из внешнего окружения и *возвращать значение* в вызвавшую среду. Предусматривается два варианта метода — с параметрами и без параметров:

```
// без параметров:
static тип  Main() { ... }
static void Main() { ... }
// с параметрами:
static тип  Main( string[] args ) { /* ... */ }
static void Main( string[] args ) { /* ... */ }
```

Параметры, разделяемые пробелами, задаются при запуске программы из командной строки после имени исполняемого файла программы. Они передаются в массив args.

ПРИМЕЧАНИЕ

Имя параметра в программе может быть любым, но принято использовать имя args.

Если метод возвращает значение, оно должно быть целого типа, если не возвращает, он должен описываться как void. В этом случае оператор возврата из Main можно опускать, а вызвавшая среда автоматически получит нулевое значение, означающее успешное завершение. Ненулевое значение обычно означает аварийное завершение, например:

```
static int Main( string[] args )
{
    ...
    if ( ... /* все пропало */ ) return 1;
    ...
    if ( ... /* абсолютно все пропало */ ) return 100;
}
```

Возвращаемое значение анализируется в командном файле, из которого запускается программа. Обычно это делается для того, чтобы можно было принять решение, выполнять ли командный файл дальше. В листинге 7.2 приводится пример метода Main, который выводит свои аргументы и ожидает нажатия любой клавиши.

Листинг 7.2. Параметры метода Main

```
using System;
namespace ConsoleApplication1
{   class Class1
```

```
    {   static void Main( string[] args )
        {
            foreach( string arg in args ) Console.WriteLine( arg );
            Console.Read();
        }
    }
}
```

Пусть исполняемый файл программы имеет имя ConsoleApplication1.exe и вызывается из командной строки:

```
d:\cs\ConsoleApplication1\bin\Debug\ConsoleApplication1.exe one two three
```

Тогда на экран будет выведено:

```
one
two
three
```

Если параметр содержит специальные символы или пробелы, его заключают в кавычки.

ПРИМЕЧАНИЕ

Для запуска программы из командной строки можно воспользоваться, к примеру, командой Выполнить меню Пуск или командой Пуск ▸ Программы ▸ Стандартные ▸ Командная строка.

Индексаторы

Индексатор представляет собой разновидность свойства. Если у класса есть скрытое поле, представляющее собой массив, то с помощью индексатора можно обратиться к элементу этого массива, используя имя объекта и номер элемента массива в квадратных скобках. Иными словами, индексатор — это такой «умный» индекс для объектов.

Синтаксис индексатора аналогичен синтаксису свойства:

```
атрибуты спецификаторы тип this [ список_параметров ]
{
    get код_доступа
    set код_доступа
}
```

ВНИМАНИЕ

В данном случае квадратные скобки являются элементом синтаксиса, а не указанием на необязательность конструкции.

Атрибуты мы рассмотрим позже, в главе 12, а *спецификаторы* аналогичны спецификаторам свойств и методов. Индексаторы чаще всего объявляются со

спецификатором public, поскольку они входят в интерфейс объекта[1]. Атрибуты и спецификаторы могут отсутствовать.

Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или установке значения (set) элемента массива. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, индексатор доступен только для чтения (read-only), если отсутствует часть get, индексатор доступен только для записи (write-only).

Список параметров содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу. Чаще всего используется один индекс целого типа.

Индексаторы в основном применяются для создания специализированных массивов, на работу с которыми накладываются какие-либо ограничения. В листинге 7.3 создан класс-массив, элементы которого должны находиться в диапазоне [0, 100]. Кроме того, при доступе к элементу проверяется, не вышел ли индекс за допустимые границы.

Листинг 7.3. Использование индексаторов

```
using System;
namespace ConsoleApplication1
{
    class SafeArray
    {   public SafeArray( int size )    // конструктор класса
        {
            a = new int[size];
            length = size;
        }
        public int Length                // свойство - размерность
        {
            get { return length; }
        }

        public int this[int i]           // индексатор
        {
            get
            {
                if ( i >= 0 && i < length ) return a[i];
                else { error = true; return 0; }
            }
            set
            {
                if ( i >= 0 && i < length &&
                    value >= 0 && value <= 100 ) a[i] = value;
```

[1] В версии C# 2.0 допускается раздельное указание спецификаторов доступа для блоков получения и установки индексатора, аналогично свойствам (раздел «Свойства», см. с. 120).

```
            else error = true;
        }
    }
    public bool error = false;      // открытый признак ошибки
    int[] a;                        // закрытый массив
    int length;                     // закрытая размерность
}

class Class1
{   static void Main()
    {
        int n = 100;
        SafeArray sa = new SafeArray( n );          // создание объекта
        for ( int i = 0; i < n; ++i )
        {
            sa[i] = i * 2;              // 1 использование индексатора
            Console.Write( sa[i] );     // 2 использование индексатора
        }
        if ( sa.error ) Console.Write( "Были ошибки!" );
    }
}
}
```

Из листинга видно, что индексаторы описываются аналогично свойствам. Благодаря применению индексаторов с объектом, заключающим в себе массив, можно работать так же, как с обычным массивом. Если обращение к объекту встречается в левой части оператора присваивания (оператор 1), автоматически вызывается метод set. Если обращение выполняется в составе выражения без побочных эффектов (оператор 2), вызывается метод get.

В классе SafeArray принята следующая стратегия обработки ошибок: если при попытке записи элемента массива его индекс или значение заданы неверно, значение элементу не присваивается; если при попытке чтения элемента индекс не входит в допустимый диапазон, возвращается 0; в обоих случаях формируется значение открытого поля error, равное true.

ПРИМЕЧАНИЕ

Проверка значения поля error может быть выполнена после какого-либо блока, предназначенного для работы с массивом, как в приведенном примере, или после каждого подозрительного действия с массивом (это может замедлить выполнение программы). Такой способ обработки ошибок не является единственно возможным. Правильнее обрабатывать подобные ошибки с помощью механизма исключений, соответствующий пример будет приведен далее в этой главе.

Вообще говоря, индексатор не обязательно должен быть связан с каким-либо внутренним полем данных. В листинге 7.4 приведен пример класса Pow2, единственное назначение которого — формировать степень числа 2.

Листинг 7.4. Индексатор без массива

```
using System;
namespace ConsoleApplication1
{
    class Pow2
    {   public ulong this[int i]
        {
            get
            {
                if ( i >= 0 )
                {
                    ulong res = 1;
                    for ( int k = 0; k < i; k++ )  // цикл получения степени
                        unchecked { res *= 2; }    // 1
                    return res;
                }
                else return 0;
            }
        }
    }

    class Class1
    {   static void Main()
        {
            int n = 13;
            Pow2 pow2 = new Pow2();
            for ( int i = 0; i < n; ++i )
                Console.WriteLine( "{0}\t{1}", i, pow2[i] );
        }
    }
}
```

Оператор 1 выполняется в непроверяемом контексте[1], для того чтобы исключение, связанное с переполнением, не генерировалось. В принципе, данная программа работает и без этого, но если поместить класс Pow2 в проверяемый контекст, при значении, превышающем допустимый диапазон для типа ulong, возникнет исключение.

Результат работы программы:

```
0       1
1       2
2       4
3       8
4       16
5       32
6       64
```

[1] Это понятие рассмотрено в разделе «Введение в исключения» (см. с. 46).

```
7       128
8       256
9       512
10      1024
11      2048
12      4096
```

Язык C# допускает использование *многомерных индексаторов*. Они описываются аналогично обычным и применяются в основном для контроля за занесением данных в многомерные массивы и выборке данных из многомерных массивов, оформленных в виде классов. Например:

```
int[,] a;
```

Если внутри класса объявлен такой двумерный массив, то заголовок индексатора должен иметь вид

```
public int this[int i, int j]
```

Операции класса

C# позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Это дает возможность применять экземпляры собственных типов данных в составе выражений таким же образом, как стандартных, например:

```
MyObject a, b, c;
...
c = a + b;                    // используется операция сложения для класса MyObject
```

Определение собственных операций класса часто называют *перегрузкой операций*. Перегрузка обычно применяется для классов, описывающих математические или физические понятия, то есть таких классов, для которых семантика операций делает программу более понятной. Если назначение операции интуитивно не понятно с первого взгляда, перегружать такую операцию не рекомендуется.

Операции класса описываются с помощью методов специального вида (*функций-операций*). Перегрузка операций похожа на перегрузку обычных методов. Синтаксис операции:

```
[ атрибуты ] спецификаторы объявитель_операции тело
```

Атрибуты рассматриваются в главе 12, в качестве *спецификаторов* одновременно используются ключевые слова `public` и `static`. Кроме того, операцию можно объявить как внешнюю (`extern`).

Объявитель операции содержит ключевое слово `operator`, по которому и опознается описание операции в классе. *Тело* операции определяет действия, которые выполняются при использовании операции в выражении. Тело представляет собой блок, аналогичный телу других методов.

ВНИМАНИЕ

Новые обозначения для собственных операций вводить нельзя. Для операций класса сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных.

При описании операций необходимо соблюдать следующие правила:

- операция должна быть описана как открытый статический метод класса (спецификаторы `public static`);
- параметры в операцию должны передаваться по значению (то есть не должны предваряться ключевыми словами `ref` или `out`);
- сигнатуры всех операций класса должны различаться;
- типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

В C# существуют три вида операций класса: унарные, бинарные и операции преобразования типа.

Унарные операции

Можно определять в классе следующие *унарные операции*:

```
+    -    !    ~    ++    --    true    false
```

Синтаксис объявителя унарной операции:

```
тип operator унарная_операция ( параметр )
```

Примеры заголовков унарных операций:

```
public static int operator +( MyObject m )
public static MyObject operator --( MyObject m )
public static bool operator true( MyObject m )
```

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция должна возвращать:

- для операций +, -, ! и ~ величину любого типа;
- для операций ++ и -- величину типа класса, для которого она определяется;
- для операций `true` и `false` величину типа `bool`.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

ПРИМЕЧАНИЕ

Префиксный и постфиксный инкременты не различаются (для них может существовать только одна реализация, которая вызывается в обоих случаях).

ПРИМЕЧАНИЕ

Операции true и false обычно определяются для логических типов SQL, обладающих неопределенным состоянием, и не входят в число тем, рассматриваемых в этой книге.

В качестве примера усовершенствуем приведенный в листинге 7.3 класс SafeArray для удобной и безопасной работы с массивом. В класс внесены следующие изменения:

- добавлен конструктор, позволяющий инициализировать массив обычным массивом или серией целочисленных значений произвольного размера;
- добавлена операция инкремента;
- добавлен вспомогательный метод Print вывода массива;
- изменена стратегия обработки ошибок выхода за границы массива;
- снято требование, чтобы элементы массива принимали значения в заданном диапазоне.

Текст программы приведен в листинге 7.5.

Листинг 7.5. Определение операции инкремента для класса SafeArray

```
using System;
namespace ConsoleApplication1
{
    class SafeArray
    {    public SafeArray( int size )                          // конструктор
        {
            a = new int[size];
            length = size;
        }

        public SafeArray( params int[] arr )            // новый конструктор
        {
            length = arr.Length;
            a = new int[length];
            for ( int i = 0; i < length; ++i ) a[i] = arr[i];
        }

        public static SafeArray operator ++( SafeArray x )              // ++
        {
            SafeArray temp = new SafeArray( x.length );
            for ( int i = 0; i < x.length; ++i )
            temp[i] = ++x.a[i];
            return temp;
        }

        public int this[int i]                                  // индексатор
        {
            get
```

продолжение

Листинг 7.5 (*продолжение*)

```
            {
                if ( i >= 0 && i < length ) return a[i];
                else throw new IndexOutOfRangeException();      // исключение
            }
            set
            {
                if ( i >= 0 && i < length ) a[i] = value;
                else throw new IndexOutOfRangeException();      // исключение
            }
        }

        public void Print( string name )                    // вывод на экран
        {
            Console.WriteLine( name + ":" );
            for ( int i = 0; i < length; ++i )
                Console.Write( "\t" + a[i] );
            Console.WriteLine();
        }
        int[] a;                        // закрытый массив
        int length;                     // закрытая размерность
    }

    class Class1
    {   static void Main()
        {
            try
            {
                SafeArray a1 = new SafeArray( 5, 2, -1, 1, -2 );
                a1.Print( "Массив 1" );
                a1++;
                a1.Print( "Инкремент массива 1" );
            }
            catch ( Exception e )                   // обработка исключения
            {
                Console.WriteLine( e.Message );
            }
        }
    }
}
```

Бинарные операции

Можно определять в классе следующие бинарные операции:

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

ВНИМАНИЕ

Операций присваивания в этом списке нет.

Синтаксис объявителя бинарной операции:

```
тип operator бинарная_операция (параметр1, параметр2)
```

Примеры заголовков бинарных операций:

```
public static MyObject operator +  ( MyObject m1, MyObject m2 )
public static bool     operator == ( MyObject m1, MyObject m2 )
```

Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа.

Операции == и !=, > и <, >= и <= определяются только парами и обычно возвращают логическое значение. Чаще всего в классе определяют операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение объектов, а не их ссылок, как определено по умолчанию для ссылочных типов. Перегрузка операций отношения требует знания интерфейсов, поэтому она рассматривается позже, в главе 9 (см. с. 203).

Пример определения операции сложения для класса SafeArray, описанного в предыдущем разделе, приведен в листинге 7.6. В зависимости от операндов операция либо выполняет поэлементное сложение двух массивов, либо прибавляет значение операнда к каждому элементу массива.

Листинг 7.6. Определение операции сложения для класса SafeArray

```
using System;
namespace ConsoleApplication1
{
    class SafeArray
    {   public SafeArray( int size )
        {
            a = new int[size];
            length = size;
        }

        public SafeArray( params int[] arr )
        {
            length = arr.Length;
            a = new int[length];
            for ( int i = 0; i < length; ++i ) a[i] = arr[i];
        }

        public static SafeArray operator + ( SafeArray x, SafeArray y ) // +
        {
            int len = x.length < y.length ? x.length : y.length;
            SafeArray temp = new SafeArray(len);

            for ( int i = 0; i < len; ++i ) temp[i] = x[i] + y[i];
            return temp;
```

продолжение

Листинг 7.6 (*продолжение*)

```
    }
    public static SafeArray operator + ( SafeArray x, int y )        // +
    {
        SafeArray temp = new SafeArray(x.length);
        for ( int i = 0; i < x.length; ++i ) temp[i] = x[i] + y;
        return temp;
    }
    public static SafeArray operator + ( int x, SafeArray y )        // +
    {
        SafeArray temp = new SafeArray(y.length);
        for ( int i = 0; i < y.length; ++i ) temp[i] = x + y[i];
        return temp;
    }
    public static SafeArray operator ++ ( SafeArray x )              // ++
    {
        SafeArray temp = new SafeArray(x.length);
        for ( int i = 0; i < x.length; ++i ) temp[i] = ++x.a[i];
        return temp;
    }

    public int this[int i]                                            // []
    {
        get
        {
            if ( i >= 0 && i < length ) return a[i];
            else throw new IndexOutOfRangeException();
        }
        set
        {
            if ( i >= 0 && i < length ) a[i] = value;
            else throw new IndexOutOfRangeException();
        }
    }

    public void Print( string name )
    {
        Console.WriteLine( name + ":" );
        for ( int i = 0; i < length; ++i ) Console.Write( "\t" + a[i] );
        Console.WriteLine();
    }

    int[] a;                          // закрытый массив
    int length;                       // закрытая размерность
}

class Class1
{   static void Main()
```

```
        {
            try
            {
                SafeArray a1 = new SafeArray( 5, 2, -1, 1, -2 );
                a1.Print( "Массив 1" );

                SafeArray a2 = new SafeArray( 1, 0, 3 );
                a2.Print( "Массив 2" ); a1++;

                SafeArray a3 = a1 + a2;
                a3.Print( "Сумма массивов 1 и 2" );

                a1 = a1 + 100;                          // 1
                a1.Print( "Массив 1 + 100" );

                a1 = 100 + a1;                          // 2
                a1.Print( "100 + массив 1" );

                a2 += ++a2 + 1;                         // 3 оторвать руки!
                a2.Print( "++a2, a2 + a2 + 1" );

            }
            catch ( Exception e )
            {
                Console.WriteLine( e.Message );
            }
        }
    }
}
```

Результат работы программы:

```
Массив 1:
        5       2       -1      1       -2
Массив 2:
        1       0       3
Сумма массивов 1 и 2:
        7       3       3
Массив 1 + 100:
        106     103     100     102     99
100 + массив 1:
        206     203     200     202     199
++a2, a2 + a2 + 1:
        5       3       9
```

Обратите внимание: чтобы обеспечить возможность сложения с константой, операция сложения перегружена два раза для случаев, когда константа является первым и вторым операндом (операторы 2 и 1).

Сложную операцию присваивания += (оператор 3) определять не требуется, да это и невозможно. При ее выполнении автоматически вызываются сначала операция

сложения, а потом присваивания. В целом же оператор 3 демонстрирует недопустимую манеру программирования, поскольку результат его выполнения неочевиден.

ПРИМЕЧАНИЕ

В перегруженных методах для объектов применяется индексатор. Для повышения эффективности можно обратиться к закрытому полю-массиву и непосредственно, например: `temp.a[i] = x + y.a[i]`.

Операции преобразования типа

Операции преобразования типа обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных. Синтаксис объявителя операции преобразования типа:

```
implicit operator тип ( параметр )      // неявное преобразование
explicit operator тип ( параметр )      // явное преобразование
```

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого определяется операция. Таким образом, операции выполняют преобразование либо типа класса к другому типу, либо наоборот. Преобразуемые типы не должны быть связаны отношениями наследования[1]. Примеры операций преобразования типа для класса Monster, описанного в главе 5:

```
public static implicit operator int( Monster m )
{
    return m.health;
}
public static explicit operator Monster( int h )
{
    return new Monster( h, 100, "FromInt" );
}
```

Ниже приведены примеры использования этих преобразований в программе. Не надо искать в них смысл, они просто иллюстрируют синтаксис:

```
Monster Masha = new Monster( 200, 200, "Masha" );
int i = Masha;                    // неявное преобразование
Masha = (Monster) 500;            // явное преобразование
```

Неявное преобразование выполняется автоматически:

- при присваивании объекта переменной целевого типа, как в примере;
- при использовании объекта в выражении, содержащем переменные целевого типа;
- при передаче объекта в метод на место параметра целевого типа;
- при явном приведении типа.

[1] Следовательно, нельзя определять преобразования к типу object и наоборот, впрочем, они уже определены без нашего участия.

Явное преобразование выполняется при использовании операции приведения типа.

Все операции класса должны иметь разные сигнатуры. В отличие от других видов методов, для операций преобразования тип возвращаемого значения включается в сигнатуру, иначе нельзя было бы определять варианты преобразования данного типа в несколько других. Ключевые слова `implicit` и `explicit` в сигнатуру не включаются, следовательно, для одного и того же преобразования нельзя определить одновременно явную и неявную версии.

Неявное преобразование следует определять так, чтобы при его выполнении не возникала потеря точности и не генерировались исключения. Если эти ситуации возможны, преобразование следует описать как явное.

Деструкторы

В C# существует специальный вид метода, называемый *деструктором*. Он вызывается *сборщиком мусора* непосредственно перед удалением объекта из памяти. В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта, например, проверяется, все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т. п.).

Синтаксис деструктора:

```
[ атрибуты ] [ extern ] ~имя_класса()
тело
```

Как видно из определения, деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется тильдой (~), символизирующей обратные по отношению к конструктору действия. *Тело* деструктора представляет собой блок или просто точку с запятой, если деструктор определен как внешний (`extern`).

Сборщик мусора удаляет объекты, на которые нет ссылок. Он работает в соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени. Поскольку деструктор вызывается сборщиком мусора, невозможно гарантировать, что деструктор будет обязательно вызван в процессе работы программы. Следовательно, его лучше использовать только для гарантии освобождения ресурсов, а «штатное» освобождение выполнять в другом месте программы.

Применение деструкторов замедляет процесс сборки мусора.

Вложенные типы

В классе можно определять типы данных, внутренние по отношению к классу. Так определяются вспомогательные типы, которые используются только содержащим их классом. Механизм вложенных типов позволяет скрыть ненужные детали

и более полно реализовать принцип инкапсуляции. Непосредственный доступ извне к такому классу невозможен (имеется в виду доступ по имени без уточнения). Для вложенных типов можно использовать те же спецификаторы, что и для полей класса[1].

Например, введем в наш класс `Monster` вспомогательный класс `Gun`. Объекты этого класса без «хозяина» бесполезны, поэтому его можно определить как внутренний:

```
using System;
namespace ConsoleApplication1
{   class Monster
    {
        class Gun
        {
            ...
        }
        ...
    }
}
```

Помимо классов вложенными могут быть и другие типы данных: интерфейсы, структуры и перечисления. Мы рассмотрим их в главе 9.

Рекомендации по программированию

Как правило, класс как тип, определенный пользователем, должен содержать скрытые (`private`) поля и следующие функциональные элементы:

- *конструкторы*, определяющие, как инициализируются объекты класса;
- набор *методов* и *свойств*, реализующих характеристики класса;
- классы *исключений*, используемые для сообщений об ошибках путем генерации исключительных ситуаций.

Классы, моделирующие математические или физические понятия, обычно также содержат набор *операций*, позволяющих копировать, присваивать, сравнивать объекты и производить с ними другие действия, требующиеся по сути класса.

Перегруженные операции класса должны иметь интуитивно понятный общепринятый смысл (например, не следует заставлять операцию + выполнять что-либо, кроме сложения или добавления). Если какая-либо операция перегружена, следует перегрузить и аналогичные операции, например + и ++ (компилятор этого автоматически не сделает). При этом операции должны иметь ту же семантику, что и их стандартные аналоги.

---

[1] Спецификаторы описаны в разделе «Данные: поля и константы» (см. с. 104).

В подавляющем большинстве классов для реализации действий с объектами класса предпочтительнее использовать не операции, а методы, поскольку им можно дать осмысленные имена.

Перегруженные методы, в отличие от операций, применяются в классах повсеместно — как минимум, используется набор перегруженных конструкторов для создания объектов различными способами.

Методы с переменным числом параметров реализуются менее эффективно, чем обычные, поэтому если, к примеру, требуется передавать в метод два, три или четыре параметра, возможно, окажется более эффективным реализовать не один метод с параметром `params`, а три перегруженных варианта с обычными параметрами.

Глава 8
Иерархии классов

Управлять большим количеством разрозненных классов довольно сложно. С этой проблемой можно справиться путем упорядочивания и ранжирования классов, то есть объединяя общие для нескольких классов свойства в одном классе и используя его в качестве базового.

Эту возможность предоставляет механизм *наследования*, который является мощнейшим инструментом ООП. Он позволяет строить иерархии, в которых классы-потомки получают свойства классов-предков и могут дополнять их или изменять. Таким образом, наследование обеспечивает важную возможность многократного использования кода. Написав и отладив код базового класса, можно, не изменяя его, за счет наследования приспособить класс для работы в различных ситуациях. Это экономит время разработки и повышает надежность программ.

Классы, расположенные ближе к началу иерархии, объединяют в себе общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных особенностей.

Итак, наследование применяется для следующих взаимосвязанных целей:

- исключения из программы повторяющихся фрагментов кода;
- упрощения модификации программы;
- упрощения создания новых программ на основе существующих.

Кроме того, наследование является единственной возможностью использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

Наследование

Класс в C# может иметь произвольное количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса

после двоеточия. Если имя предка не указано, предком считается базовый класс всей иерархии System.Object:

```
[ атрибуты ] [ спецификаторы ] class имя_класса [ : предки ]
    тело класса
```

ПРИМЕЧАНИЕ

Обратите внимание на то, что слово «предки» присутствует в описании класса во множественном числе, хотя класс может иметь только одного предка. Причина в том, что класс наряду с единственным предком может наследовать от интерфейсов — специального вида классов, не имеющих реализации. Интерфейсы рассматриваются в следующей главе.

Рассмотрим наследование классов на примере. В разделе «Свойства» (см. с. 120) был описан класс Monster, моделирующий персонаж компьютерной игры. Допустим, нам требуется ввести в игру еще один тип персонажей, который должен обладать свойствами объекта Monster, а кроме того уметь думать. Будет логично сделать новый объект потомком объекта Monster (листинг 8.1).

Листинг 8.1. Класс Daemon, потомок класса Monster

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        ...
    }

    class Daemon : Monster
    {
        public Daemon()
        {
            brain = 1;
        }

        public Daemon( string name, int brain ) : base( name )          // 1
        {
            this.brain = brain;
        }

        public Daemon( int health, int ammo, string name, int brain )
            : base( health, ammo, name )                                // 2
        {
            this.brain = brain;
        }

        new public void Passport()                                      // 3
```

продолжение

Листинг 8.1 (*продолжение*)

```
        {
            Console.WriteLine(
                "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
                Name, Health, Ammo, brain );
        }

        public void Think()                                          // 4
        {
            Console.Write( Name + " is" );
            for ( int i = 0; i < brain; ++i ) Console.Write( " thinking" );
            Console.WriteLine( "..." );
        }

        int brain;        // закрытое поле
    }

    class Class1
    {   static void Main()
        {
            Daemon Dima = new Daemon( "Dima", 3 );                   // 5
            Dima.Passport();                                         // 6
            Dima.Think();                                            // 7
            Dima.Health -= 10;                                       // 8
            Dima.Passport();
        }
    }
}
```

В классе Daemon введены закрытое поле brain и метод Think, определены собственные конструкторы, а также переопределен метод Passport. Все поля и свойства класса Monster наследуются в классе Daemon[1].

Результат работы программы:

```
Daemon Dima      health = 100 ammo = 100 brain = 3
Dima is thinking thinking thinking...
Daemon Dima      health = 90 ammo = 100 brain = 3
```

Как видите, экземпляр класса Daemon с одинаковой легкостью использует как собственные (операторы 5–7), так и унаследованные (оператор 8) элементы класса. Рассмотрим общие правила наследования, используя в качестве примера листинг 8.1.

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными далее правилами:

[1] Если бы в классе Monster были описаны методы, не переопределенные в Daemon, они бы также были унаследованы.

- Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров. Это правило использовано в первом из конструкторов класса Daemon.
- Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким образом, каждый конструктор инициализирует свою часть объекта.
- Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации (это продемонстрировано в конструкторах, вызываемых в операторах 1 и 2). Вызов выполняется с помощью ключевого слова base. Вызывается та версия конструктора, список параметров которой соответствует списку аргументов, указанных после слова base.

Поля, *методы* и *свойства* класса наследуются, поэтому при желании заменить элемент базового класса новым элементом следует явным образом указать компилятору свое намерение с помощью ключевого слова new[1]. В листинге 8.1 таким образом переопределен метод вывода информации об объекте Passport. Другой способ переопределения методов рассматривается далее в разделе «Виртуальные методы».

Метод Passport класса Daemon замещает соответствующий метод базового класса, однако возможность доступа к методу базового класса из метода производного класса сохраняется. Для этого перед вызовом метода указывается все то же волшебное слово base, например:

```
base.Passport();
```

СОВЕТ

Вызов одноименного метода предка из метода потомка всегда позволяет сохранить функции предка и дополнить их, не повторяя фрагмент кода. Помимо уменьшения объема программы это облегчает ее модификацию, поскольку изменения, внесенные в метод предка, автоматически отражаются во всех его потомках. В конструкторах метод предка вызывается после списка параметров и двоеточия, а в остальных методах — в любом месте с помощью приведенного синтаксиса.

Вот, например, как выглядел бы метод Passport, если бы мы в классе Daemon хотели не полностью переопределить поведение его предка, а дополнить его:

```
new public void Passport()
{
    base.Passport();
    Console.WriteLine( " brain = {1}", brain );
}
```

[1] Можно этого не делать, но тогда компилятор выдаст предупреждение. Предупреждение (warning) — это не ошибка, оно не препятствует успешной компиляции, но тем не менее...

Элементы базового класса, определенные как private, в производном классе недоступны. Поэтому в методе Passport для доступа к полям name, health и ammo пришлось использовать соответствующие свойства базового класса. Другое решение заключается в том, чтобы определить эти поля со спецификатором protected, в этом случае они будут доступны методам всех классов, производных от Monster. Оба решения имеют свои достоинства и недостатки.

ВНИМАНИЕ

Важно понимать, что на этапе выполнения программы объект представляет собой единое целое, не разделенное на части предка и потомка.

Во время выполнения программы объекты хранятся в отдельных переменных, массивах или других коллекциях[1]. Во многих случаях удобно *оперировать объектами одной иерархии единообразно*, то есть использовать один и тот же программный код для работы с экземплярами разных классов. Желательно иметь возможность описать:

- объект, в который во время выполнения программы заносятся ссылки на объекты разных классов иерархии;
- контейнер, в котором хранятся объекты разных классов, относящиеся к одной иерархии;
- метод, в который могут передаваться объекты разных классов иерархии;
- метод, из которого в зависимости от типа вызвавшего его объекта вызываются соответствующие методы.

Все это возможно благодаря тому, что *объекту базового класса можно присвоить объект производного класса*[2] .

Давайте попробуем описать массив объектов базового класса и занести туда объекты производного класса. В листинге 8.2 в массиве типа Monster хранятся два объекта типа Monster и один — типа Daemon.

Листинг 8.2. Массив объектов разных типов

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        ...
    }

    class Daemon : Monster
```

[1] *Коллекция* — объект, предназначенный для хранения данных и предоставляющий методы доступа к ним. Например, массив предоставляет прямой доступ к любому его элементу по индексу. Коллекции библиотеки .NET рассматриваются в главе 13.

[2] Еще раз напомню, что объекты относятся к ссылочному типу, следовательно, присваивается не сам объект, а ссылка на него.

```
    {
        ... // см. листинг 8.1
    }

    class Class1
    {   static void Main()
        {
            const int n = 3;
            Monster[] stado = new Monster[n];

            stado[0] = new Monster( "Monia" );
            stado[1] = new Monster( "Monk" );
            stado[2] = new Daemon ( "Dimon", 3 );

            foreach ( Monster elem in stado ) elem.Passport();          // 1

            for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;            // 2
            Console.WriteLine();

            foreach ( Monster elem in stado ) elem.Passport();          // 3
        }
    }
}
```

Результат работы программы:

```
Monster Monia    health = 100 ammo = 100
Monster Monk     health = 100 ammo = 100
Monster Dimon    health = 100 ammo = 100

Monster Monia    health = 100 ammo = 0
Monster Monk     health = 100 ammo = 0
Monster Dimon    health = 100 ammo = 0
```

Результат радует нас только частично: объект типа `Daemon` действительно можно поместить в массив, состоящий из элементов типа `Monster`, но для него вызываются только методы и свойства, унаследованные от предка. Это устраивает нас в операторе 2, а в операторах 1 и 3 хотелось бы, чтобы вызывался метод `Passport`, переопределенный в потомке.

Итак, присваивать объекту базового класса объект производного класса можно, но вызываются для него только методы и свойства, определенные в базовом классе. Иными словами, возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает.

Это и понятно: ведь компилятор должен еще до выполнения программы решить, какой метод вызывать, и вставить в код фрагмент, передающий управление на этот метод (этот процесс называется *ранним связыванием*). При этом компилятор может руководствоваться только типом переменной, для которой вызывается метод или свойство (например, `stado[i].Ammo`). То, что в этой переменной в разные

моменты времени могут находиться ссылки на объекты разных типов, компилятор учесть не может.

Следовательно, если мы хотим, чтобы вызываемые методы соответствовали типу объекта, необходимо отложить процесс связывания до этапа выполнения программы, а точнее — до момента вызова метода, когда уже точно известно, на объект какого типа указывает ссылка. Такой механизм в C# есть — он называется *поздним связыванием* и реализуется с помощью так называемых виртуальных методов, которые мы незамедлительно и рассмотрим.

Виртуальные методы

При раннем связывании программа, готовая для выполнения, представляет собой структуру, логика выполнения которой жестко определена. Если же требуется, чтобы решение о том, какой из одноименных методов разных объектов иерархии использовать, принималось в зависимости от конкретного объекта, для которого выполняется вызов, то заранее жестко связывать эти методы с остальной частью кода нельзя.

Следовательно, надо каким-то образом дать знать компилятору, что эти методы будут обрабатываться по-другому. Для этого в C# существует ключевое слово `virtual`. Оно записывается в заголовке метода базового класса, например:

```
virtual public void Passport() ...
```

Слово `virtual` в переводе с английского значит «фактический». Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться по факту его вызова, то есть не на стадии компиляции, а во время выполнения программы. Этот механизм называется *поздним связыванием*.

Для его реализации необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет в любой момент воспользоваться, поэтому компилятор формирует для этих методов *таблицу виртуальных методов* (Virtual Method Table, VMT). В нее записываются адреса виртуальных методов (в том числе унаследованных) в порядке описания в классе. Для каждого класса создается одна таблица.

Каждый объект во время выполнения должен иметь доступ к VMT. Обеспечение этой связи нельзя поручить компилятору, так как она должна устанавливаться во время выполнения программы при создании объекта. Поэтому связь экземпляра объекта с VMT устанавливается с помощью специального кода, автоматически помещаемого компилятором в конструктор объекта.

Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово `override`, например:

```
override public void Passport() ...
```

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса. Это требование вполне естественно, если учесть, что одноименные методы, относящиеся к разным классам, могут вызываться из одной и той же точки программы.

Добавим в листинг 8.2 два волшебных слова — virtual и override — в описания методов Passport, соответственно, базового и производного классов (листинг 8.3).

Листинг 8.3. Виртуальные методы

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        ...
        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }
        ...
    }

    class Daemon : Monster
    {
        ...
        override public void Passport()
        {
            Console.WriteLine(
                "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
                Name, Health, Ammo, brain );
        }
        ...
    }

    class Class1
    {   static void Main()
        {
            const int n = 3;
            Monster[] stado = new Monster[n];

            stado[0] = new Monster( "Monia" );
            stado[1] = new Monster( "Monk" );
            stado[2] = new Daemon ( "Dimon", 3 );

            foreach ( Monster elem in stado ) elem.Passport();

            for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;
            Console.WriteLine();

            foreach ( Monster elem in stado ) elem.Passport();
        }
    }
}
```

Результат работы программы:

```
Monster Monia    health = 100 ammo = 100
Monster Monk     health = 100 ammo = 100
Daemon Dimon     health = 100 ammo = 100 brain = 3

Monster Monia    health = 100 ammo = 0
Monster Monk     health = 100 ammo = 0
Daemon Dimon     health = 100 ammo = 0 brain = 3
```

Как видите, теперь в циклах 1 и 3 вызывается метод `Passport`, соответствующий типу объекта, помещенного в массив.

Виртуальные методы базового класса определяют интерфейс всей иерархии. Этот интерфейс может расширяться в потомках за счет добавления новых виртуальных методов. Переопределять виртуальный метод в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, метод наследуется.

Вызов виртуального метода выполняется так: из объекта берется адрес его таблицы VMT, из VMT выбирается адрес метода, а затем управление передается этому методу. Таким образом, при использовании виртуальных методов из всех одноименных методов иерархии всегда выбирается тот, который соответствует фактическому типу вызвавшего его объекта.

ПРИМЕЧАНИЕ

Вызов виртуального метода, в отличие от обычного, выполняется через дополнительный этап получения адреса метода из таблицы VMT, что несколько замедляет выполнение программы.

С помощью виртуальных методов реализуется один из основных принципов объектно-ориентированного программирования — *полиморфизм*. Это слово в переводе с греческого означает «много форм», что в данном случае означает «один вызов — много методов». Применение виртуальных методов обеспечивает гибкость и возможность расширения функциональности класса.

Виртуальные методы незаменимы и при передаче объектов в методы в качестве параметров. В параметрах метода описывается объект базового типа, а при вызове в него передается объект производного класса. В этом случае виртуальные методы, вызываемые для объекта из метода, будут соответствовать типу аргумента, а не параметра.

При описании классов рекомендуется определять в качестве виртуальных те методы, которые в производных классах должны реализовываться по-другому. Если во всех классах иерархии метод будет выполняться одинаково, его лучше определить как обычный метод.

ПРИМЕЧАНИЕ

Все сказанное о виртуальных методах относится также к свойствам и индексаторам.

Абстрактные классы

При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский класс. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что никакие реальные объекты им не соответствуют. Такие классы называют абстрактными.

Абстрактный класс служит только для порождения потомков. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по-своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.

Абстрактный класс задает интерфейс для всей иерархии, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором `abstract`.

ПРИМЕЧАНИЕ

Абстрактный класс может содержать и полностью определенные методы, в отличие от сходного с ним по предназначению специального вида класса, называемого интерфейсом. Интерфейсы рассматриваются в следующей главе.

Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть описан как абстрактный, например:

```
abstract class Spirit
{
    public abstract void Passport();
}
class Monster : Spirit
{
    ...
    override public void Passport()
    {
        Console.WriteLine( "Monster.{0} \t health = {1} ammo = {2}",
                           name, health, ammo );
    }
    ...
}

class Daemon : Monster
{
    ...
    override public void Passport()
    {
        Console.WriteLine(
            "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
            Name, Health, Ammo, brain );
    }
    ... // полный текст этих классов приведен в главе 12
}
```

Абстрактные классы используются при работе со структурами данных, предназначенными для хранения объектов одной иерархии, и в качестве параметров методов. Если класс, производный от абстрактного, не переопределяет все абстрактные методы, он также должен описываться как абстрактный.

Можно создать метод, параметром которого является абстрактный класс. На место этого параметра при выполнении программы может передаваться объект любого производного класса. Это позволяет создавать *полиморфные методы*, работающие с объектом любого типа в пределах одной иерархии. Полиморфизм в различных формах является мощным и широко применяемым инструментом ООП.

ПРИМЕЧАНИЕ

Мы уже использовали полиморфизм в разделе «Оператор foreach» (см. с. 136) для того, чтобы метод `PrintArray` мог работать с массивом любого типа. Еще один пример применения абстрактных и виртуальных методов имеется в главе 10.

Бесплодные классы

В C# есть ключевое слово `sealed`, позволяющее описать класс, от которого, в противоположность абстрактному, наследовать запрещается:

```
    sealed class Spirit
    {
        ...
    }
//  class Monster : Spirit { ... }          ошибка!
```

Большинство встроенных типов данных описано как `sealed`. Если необходимо использовать функциональность бесплодного класса, применяется не наследование, а *вложение*, или *включение*: в классе описывается поле соответствующего типа.

Вложение классов, когда один класс включает в себя поля, являющиеся классами, является *альтернативой наследованию* при проектировании. Например, если есть объект «двигатель», а требуется описать объект «самолет», логично сделать двигатель полем этого объекта, а не его предком.

Поскольку поля класса обычно закрыты, возникает вопрос, как же пользоваться методами включенного объекта. Общепринятый способ состоит в том, чтобы описать метод объемлющего класса, из которого вызвать метод включенного класса. Такой способ взаимоотношений классов известен как *модель включения-делегирования*. Пример приведен в листинге 8.4.

Листинг 8.4. Модель включения-делегирования

```
using System;
namespace ConsoleApplication1
{
    class Двигатель
    {   public void Запуск()
```

```
        {
            Console.WriteLine( "вжжжж!!" );
        }
    }
    class Самолет
    {  public Самолет()
       {
            левый  = new Двигатель();
            правый = new Двигатель();
        }
        public void Запустить_двигатели()
        {
            левый.Запуск();
            правый.Запуск();
        }
        Двигатель левый, правый;
    }

    class Class1
    {   static void Main()
        {
        Самолет АН24_1 = new Самолет();
        АН24_1.Запустить_двигатели();
        }
    }
}
```

Результат работы программы:

```
вжжжж!!
вжжжж!!
```

В методе `Запустить_двигатели` запрос на запуск двигателей передается, или, как принято говорить, *делегируется* вложенному классу.

В отличие от наследования, когда производный класс «является» (*is a*) разновидностью базового, модель включения-делегирования реализует отношение «имеет» (*has a*). При проектировании классов следует выбирать модель, наиболее точно отражающую смысл взаимоотношений классов, например, моделируемых объектов предметной области.

Класс object

Корневой класс `System.Object` всей иерархии объектов .NET, называемый в C# `object`, обеспечивает всех наследников несколькими важными методами. Производные классы могут использовать эти методы непосредственно или переопределять их.

Класс object часто используется и непосредственно при описании типа параметров методов для придания им общности, а также для хранения ссылок на объекты различного типа — таким образом реализуется полиморфизм.

Открытые методы класса System.Object перечислены ниже.

- Метод Equals с одним параметром возвращает значение true, если параметр и вызывающий объект ссылаются на одну и ту же область памяти. Синтаксис:

```
public virtual bool Equals( object obj );
```

- Метод Equals с двумя параметрами возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

```
public static bool Equals( object ob1, object ob2 );
```

- Метод GetHashCode формирует хеш-код объекта и возвращает число, однозначно идентифицирующее объект. Это число используется в различных структурах и алгоритмах библиотеки. Если переопределяется метод Equals, необходимо перегрузить и метод GetHashCode. Подробнее о хеш-кодах рассказывается в разделе «Абстрактные структуры данных» (см. с. 291). Синтаксис:

```
public virtual int GetHashCode();
```

- Метод GetType возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает. Возвращаемое значение имеет тип Type. Это абстрактный базовый класс иерархии, использующийся для получения информации о типах во время выполнения[1]. Синтаксис:

```
public Type GetType();
```

- Метод ReferenceEquals возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

```
public static bool ReferenceEquals( object ob1, object ob2 );
```

- Метод ToString по умолчанию возвращает для ссылочных типов полное имя класса в виде строки, а для значимых — значение величины, преобразованное в строку. Этот метод переопределяют для того, чтобы можно было выводить информацию о состоянии объекта. Синтаксис:

```
public virtual string ToString()
```

В производных объектах эти методы часто переопределяют. Например, можно переопределить метод Equals для того, чтобы задать собственные критерии сравнения объектов, потому что часто бывает удобнее использовать для сравнения не ссылочную семантику (равенство ссылок), а значимую (равенство значений).

Пример применения и переопределения методов класса object для класса Monster приведен в листинге 8.5.

---

[1] Мы рассмотрим этот тип в главе 12 (см. с. 279).

Листинг 8.5. Перегрузка методов класса object

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }

        public override bool Equals( object obj )
        {
            if ( obj == null || GetType() != obj.GetType() ) return false;

            Monster temp = (Monster) obj;
            return  health == temp.health &&
                    ammo   == temp.ammo   &&
                    name   == temp.name;
        }

        public override int GetHashCode()
        {
            return name.GetHashCode();
        }

        public override string ToString()
        {
            return string.Format( "Monster {0} \t health = {1} ammo = {2}",
                    name, health, ammo );
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            Monster X = new Monster( 80, 80, "Вася" );
            Monster Y = new Monster( 80, 80, "Вася" );
            Monster Z = X;

            if ( X == Y ) Console.WriteLine(" X == Y ");
            else          Console.WriteLine(" X != Y ");
```

продолжение

Листинг 8.5 (*продолжение*)

```
            if ( X == Z ) Console.WriteLine(" X == Z ");
            else          Console.WriteLine(" X != Z ");

            if ( X.Equals(Y) ) Console.WriteLine( " X Equals Y " );
            else               Console.WriteLine( " X not Equals Y " );

            Console.WriteLine(X.GetType());
        }
    }
}
```

Результат работы программы:

```
X != Y
X == Z
X Equals Y
ConsoleApplication1.Monster
```

В методе `Equals` сначала проверяется переданный в него аргумент. Если он равен `null` или его тип не соответствует типу объекта, вызвавшего метод, возвращается значение `false`. Значение `true` формируется в случае попарного равенства всех полей объектов.

Метод `GetHashCode` просто делегирует свои функции соответствующему методу одного из полей. Метод `ToString` формирует форматированную строку, содержащую значения полей объекта.

Анализируя результат работы программы, можно увидеть, что в операции сравнения на равенство сравниваются ссылки, а в перегруженном методе `Equals` — значения. Для концептуального единства можно переопределить и операции отношения, соответствующий пример приведен в разделе «Перегрузка операций отношения» (см. с. 203).

Рекомендации по программированию

Наследование классов предоставляет программисту богатейшие возможности организации кода и его многократного использования. Выбор наиболее подходящих средств для целей конкретного проекта основывается на знании механизма их работы и взаимодействия.

Наследование класса `Y` от класса `X` означает, что `Y` представляет собой разновидность класса `X`, то есть более конкретную, частную концепцию. Базовый класс `X` является более общим понятием, чем `Y`[1]. Везде, где можно использовать `X`, можно использовать и `Y`, но не наоборот (вспомните, что на место базового класса можно передавать любой из производных). Необходимо помнить, что во время

[1] Например, каждый программист — человек, но не каждый человек — программист.

выполнения программы не существует иерархии классов и передачи сообщений объектам базового класса из производных — есть только конкретные объекты классов, поля которых формируются на основе иерархии на этапе компиляции.

Главное преимущество наследования состоит в том, что на уровне базового класса можно написать универсальный код, с помощью которого работать также с объектами производного класса, что реализуется с помощью виртуальных методов.

Как виртуальные должны быть описаны методы, которые выполняют во всех классах иерархии одну и ту же функцию, но, возможно, разными способами. Пусть, например, все объекты иерархии должны уметь выводить информацию о себе. Поскольку эта информация хранится в различных полях производных классов, функцию вывода нельзя реализовать в базовом классе. Естественно назвать ее во всех классах одинаково и объявить как виртуальную с тем, чтобы ее можно было вызывать в зависимости от фактического типа объекта, с которым работают через базовый класс.

Для представления общих понятий, которые предполагается конкретизировать в производных классах, используют абстрактные классы. Как правило, в абстрактном классе задается набор методов, то есть интерфейс, который каждый из потомков будет реализовывать по-своему.

Обычные (не виртуальные) методы переопределять в производных классах не рекомендуется, поскольку производные классы должны наследовать свойства базовых, а спецификатор `new`, с помощью которого переопределяется обычный метод, «разрывает» отношение наследования на уровне метода. Иными словами, невиртуальный метод должен быть инвариантен относительно специализации, то есть должен сохранять свойства, унаследованные из базового класса независимо от того, как конкретизируется (специализируется) производный класс. Специализация производного класса достигается добавлением новых методов и переопределением существующих виртуальных методов.

Альтернативным наследованию механизмом использования одним классом другого является вложение, когда один класс является полем другого. *Вложение* представляет отношения классов «`Y` содержит `X`» или «`Y` реализуется посредством `X`».

Для выбора между наследованием и вложением служит ответ на вопрос о том, может ли у `Y` быть несколько объектов класса `X` («`Y` содержит `X`»). Кроме того, вложение используется вместо наследования тогда, когда про классы `X` и `Y` нельзя сказать, что `Y` является разновидностью `X`, но при этом `Y` использует часть функциональности `X` («`Y` реализуется посредством `X`»).

Глава 9
Интерфейсы и структурные типы

В этой главе рассматриваются специальные виды классов — интерфейсы, структуры и перечисления.

Синтаксис интерфейса

Интерфейс является «крайним случаем» абстрактного класса. В нем задается набор абстрактных методов, свойств и индексаторов, которые должны быть реализованы в производных классах[1]. Иными словами, интерфейс определяет поведение, которое поддерживается реализующими этот интерфейс классами. Основная идея использования интерфейса состоит в том, чтобы к объектам таких классов можно было обращаться одинаковым образом.

Каждый класс может определять элементы интерфейса по-своему. Так достигается полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода.

Синтаксис интерфейса аналогичен синтаксису класса:

```
[ атрибуты ] [ спецификаторы ] interface имя_интерфейса [ : предки ]
    тело_интерфейса [ ; ]
```

Для интерфейса могут быть указаны *спецификаторы* `new`, `public`, `protected`, `internal` и `private`. Спецификатор `new` применяется для вложенных интерфейсов и имеет такой же смысл, как и соответствующий модификатор метода класса. Остальные спецификаторы управляют видимостью интерфейса. В разных контекстах определения интерфейса допускаются разные спецификаторы. По умолчанию интерфейс доступен только из сборки, в которой он описан (`internal`).

---

[1] Кроме того, в интерфейсе можно описывать события, которые мы рассмотрим в главе 10.

Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае *предки* перечисляются через запятую. *Тело интерфейса* составляют абстрактные методы, шаблоны свойств и индексаторов, а также события.

ПРИМЕЧАНИЕ

Методом исключения можно догадаться, что интерфейс не может содержать константы, поля, операции, конструкторы, деструкторы, типы и любые статические элементы.

В качестве примера рассмотрим интерфейс `IAction`, определяющий базовое поведение персонажей компьютерной игры, встречавшихся в предыдущих главах. Допустим, что любой персонаж должен уметь выводить себя на экран, атаковать и красиво умирать:

```
interface IAction
{
    void Draw();
    int Attack(int a);
    void Die();
    int Power { get; }
}
```

В интерфейсе `IAction` заданы заголовки трех методов и шаблон свойства `Power`, доступного только для чтения. Как легко догадаться, если бы требовалось обеспечить еще и возможность установки свойства, в шаблоне следовало указать ключевое слово `set`, например:

```
int Power { get; set; }
```

В интерфейсе имеет смысл задавать заголовки тех методов и свойств, которые будут по-разному реализованы различными классами разных иерархий.

ВНИМАНИЕ

Если некий набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии. То, что работает в пределах иерархии одинаково, предпочтительно полностью определить в базовом классе (примерами таких действий являются свойства `Health`, `Ammo` и `Name` из иерархии персонажей игры). Интерфейсы же чаще используются для задания общих свойств объектов различных иерархий.

Отличия интерфейса от абстрактного класса:

- элементы интерфейса по умолчанию имеют спецификатор доступа `public` и не могут иметь спецификаторов, заданных явным образом;
- интерфейс не может содержать полей и обычных методов — все элементы интерфейса должны быть абстрактными;

- класс, в списке предков которого задается интерфейс, должен определять *все* его элементы, в то время как потомок абстрактного класса может не переопределять часть абстрактных методов предка (в этом случае производный класс также будет абстрактным);
- класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

ПРИМЕЧАНИЕ

В библиотеке .NET определено большое количество стандартных интерфейсов, которые описывают поведение объектов разных классов. Например, если требуется сравнивать объекты по принципу больше или меньше, соответствующие классы должны реализовать интерфейс `IComparable`. Мы рассмотрим наиболее употребительные стандартные интерфейсы в последующих разделах этой главы.

Реализация интерфейса

В списке предков класса сначала указывается его базовый класс, если он есть, а затем через запятую — интерфейсы, которые реализует этот класс. Таким образом, в C# поддерживается одиночное наследование для классов и множественное — для интерфейсов. Это позволяет придать производному классу свойства нескольких базовых интерфейсов, реализуя их по своему усмотрению.

Например, реализация интерфейса `IAction` в классе `Monster` может выглядеть следующим образом:

```
using System;
namespace ConsoleApplication1
{
    interface IAction
    {
        void Draw();
        int Attack( int a );
        void Die();
        int Power { get; }
    }

    class Monster : IAction
    {
        public void Draw()
        {
            Console.WriteLine( "Здесь был " + name );

        }
        public int Attack( int ammo_ )
        {
            ammo -= ammo_;
            if ( ammo > 0 ) Console.WriteLine( "Ба-бах!" );
            else            ammo = 0;
```

```
        return ammo;
    }

    public void Die()
    {
        Console.WriteLine( "Monster " + name + " RIP" );
        health = 0;
    }

    public int Power
    {
        get
        {
            return ammo * health;
        }
    }

    ...
}
```

Естественно, что сигнатуры методов в интерфейсе и реализации должны полностью совпадать. Для реализуемых элементов интерфейса в классе следует указывать спецификатор public. К этим элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса[1]:

```
Monster Vasia = new Monster( 50, 50, "Вася" );  // объект класса Monster
Vasia.Draw();                                   // результат: Здесь был Вася
IAction Actor = new Monster( 10, 10, "Маша" );  // объект типа интерфейса
Actor.Draw();                                   // результат: Здесь был Маша
```

Удобство второго способа проявляется при присваивании объектам типа IAction ссылок на объекты различных классов, поддерживающих этот интерфейс. Например, легко себе представить метод с параметром типа интерфейса. На место этого параметра можно передавать любой объект, реализующий интерфейс:

```
static void Act( IAction A )
{
    A.Draw();
}

static void Main()
{
    Monster Vasia = new Monster( 50, 50, "Вася" );
    Act( Vasia );
    ...
}
```

[1] Этот объект должен содержать ссылку на класс, поддерживающий интерфейс. Естественно, что объекты типа интерфейса, так же как и объекты абстрактных классов, создавать нельзя.

Существует второй способ реализации интерфейса в классе: *явное указание имени интерфейса* перед реализуемым элементом. Спецификаторы доступа при этом не указываются. К таким элементам можно обращаться в программе *только через объект типа интерфейса*, например:

```
    class Monster : IAction
    {
        int IAction.Power
        {
            get
            {
                return ammo * health;
            }
        }

        void IAction.Draw()
        {
            Console.WriteLine( "Здесь был " + name );
        }
        ...
    }
...
IAction Actor = new Monster( 10, 10, "Маша" );
Actor.Draw();                          // обращение через объект типа интерфейса

// Monster Vasia = new Monster( 50, 50, "Вася" );
// Vasia.Draw();                                                    ошибка!
```

Таким образом, при явном задании имени реализуемого интерфейса соответствующий метод *не входит в интерфейс класса*. Это позволяет упростить его в том случае, если какие-то элементы интерфейса не требуются конечному пользователю класса.

Кроме того, явное задание имени реализуемого интерфейса перед именем метода позволяет избежать конфликтов при множественном наследовании, если элементы с одинаковыми именами или сигнатурой встречаются более чем в одном интерфейсе[1]. Пусть, например, класс Monster поддерживает два интерфейса: один для управления объектами, а другой для тестирования:

```
interface ITest
{
    void Draw();
}

interface IAction
```

[1] Методы с одинаковыми именами, но с различными сигнатурами к конфликту не приводят, они просто считаются перегруженными.

```
{
    void Draw();
    int Attack( int a );
    void Die();
    int Power { get; }
}

class Monster : IAction, ITest
{
    void ITest.Draw()
    {
        Console.WriteLine( "Testing " + name );
    }

    void IAction.Draw()
    {
        Console.WriteLine( "Здесь был " + name );
    }
    ...
}
```

Оба интерфейса содержат метод Draw с одной и той же сигнатурой. Различать их помогает явное указание имени интерфейса. Обращаться к этим методам можно, используя операцию приведения типа, например:

```
Monster Vasia = new Monster( 50, 50, "Вася" );
((ITest)Vasia).Draw();                    // результат: Здесь был Вася
((IAction)Vasia).Draw();                  // результат:   Testing Вася
```

Впрочем, если от таких методов не требуется разное поведение, можно реализовать метод первым способом (со спецификатором public), компилятор не возражает:

```
class Monster : IAction, ITest
{
    public void Draw()
    {
        Console.WriteLine( "Здесь был " + name );
    }
    ...
}
```

К методу Draw, описанному таким образом, можно обращаться любым способом: через объект класса Monster, через интерфейс IAction или ITest.

Конфликт возникает в том случае, если компилятор не может определить из контекста обращения к элементу, элемент какого именно из реализуемых интерфейсов требуется вызвать. При этом всегда помогает явное задание имени интерфейса.

Работа с объектами через интерфейсы. Операции is и as

При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс. Проверка выполняется с помощью бинарной операции `is`. Эта операция определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова `is`, с типом, заданным справа.

Результат операции равен `true`, если объект можно преобразовать к заданному типу, и `false` в противном случае. Операция обычно используется в следующем контексте:

```
if ( объект is тип )
{
    // выполнить преобразование "объекта" к "типу"
    // выполнить действия с преобразованным объектом
}
```

Допустим, мы оформили какие-то действия с объектами в виде метода с параметром типа `object`. Прежде чем использовать этот параметр внутри метода для обращения к методам, описанным в производных классах, требуется выполнить преобразование к производному классу. Для безопасного преобразования следует проверить, возможно ли оно, например так:

```
static void Act( object A )
{
    if ( A is IAction )
    {
        IAction Actor = (IAction) A;
        Actor.Draw();
    }
}
```

В метод `Act` можно передавать любые объекты, но на экран будут выведены только те, которые поддерживают интерфейс `IAction`.

Недостатком использования операции `is` является то, что преобразование фактически выполняется дважды: при проверке и при собственно преобразовании. Более эффективной является другая операция — `as`. Она выполняет преобразование к заданному типу, а если это невозможно, формирует результат `null`, например:

```
static void Act( object A )
{
    IAction Actor = A as IAction;
    if ( Actor != null ) Actor.Draw();
}
```

Обе рассмотренные операции применяются как к интерфейсам, так и к классам.

Интерфейсы и наследование

Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в последнем случае он наследует все элементы всех своих базовых интерфейсов, начиная с самого верхнего уровня. Базовые интерфейсы должны быть доступны в не меньшей степени, чем их потомки. Например, нельзя использовать интерфейс, описанный со спецификатором `private` или `internal`, в качестве базового для открытого (`public`) интерфейса[1].

Как и в обычной иерархии классов, базовые интерфейсы определяют общее поведение, а их потомки конкретизируют и дополняют его. В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово `new`, как и в аналогичной ситуации в классах. С помощью этого слова соответствующий элемент базового интерфейса скрывается. Вот пример из документации C#:

```
interface IBase
{
   void F( int i );
}
interface Ileft : IBase
{
   new void F( int i );                  // переопределение метода F
}
interface Iright : IBase
{
   void G();
}
interface Iderived : ILeft, IRight {}

class A
{
   void Test( IDerived d ) {
      d.F( 1 );                          // Вызывается ILeft.F
      ((IBase)d).F( 1 );                 // Вызывается IBase.F
      ((ILeft)d).F( 1 );                 // Вызывается ILeft.F
      ((IRight)d).F( 1 );                // Вызывается IBase.F
   }
}
```

Метод `F` из интерфейса `IBase` скрыт интерфейсом `ILeft`, несмотря на то что в цепочке `IDerived` — `IRight` — `IBase` он не переопределялся.

Класс, реализующий интерфейс, должен определять все его элементы, в том числе унаследованные. Если при этом явно указывается имя интерфейса, оно

[1] Естественно, интерфейс не может быть наследником самого себя.

должно ссылаться на тот интерфейс, в котором был описан соответствующий элемент, например:

```
class A : IRight
{
    IRight.G() { ... }
    IBase.F( int i ) { ... }        // IRight.F( int i ) - нельзя
}
```

Интерфейс, на собственные или унаследованные элементы которого имеется явная ссылка, должен быть указан в списке предков класса, например:

```
class B : A
{
//    IRight.G() { ... }             нельзя!
}
class C : A, IRight
{
    IRight.G() { ... }             // можно
    IBase.F( int i ) { ... }       // можно
}
```

Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора new, но обращаться к ним можно будет только через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия:

```
interface IBase
{
   void A();
}

class Base : IBase
{
   public void A() { ... }
}

class Derived: Base
{
   new public void A() { ... }
}

...
Derived d = new Derived ();
d.A();                          // вызывается Derived.A();
IBase id = d;
id.A();                         // вызывается Base.A();
```

Однако если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату:

```
interface IBase
{
   void A();
}

class Base : IBase
{
   public virtual void A() { ... }
}

class Derived: Base
{
   public override void A() { ... }
}

...
Derived d = new Derived ();
d.A();                          // вызывается Derived.A();
IBase id = d;
id.A();                         // вызывается Derived.A();
```

Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается. При необходимости переопределить в потомках его поведение пользуются следующим приемом: из этого метода вызывается другой, защищенный метод, который объявляется виртуальным. В приведенном далее примере метод A интерфейса IBase реализуется посредством защищенного виртуального метода A_, который можно переопределять в потомках класса Base:

```
interface IBase
{
   void A();
}

class Base : IBase
{
   void IBase.A() { A_(); }
   protected virtual void A_() { ... }
}

class Derived: Base
{
   protected override void A_() { ... }
}
```

Существует возможность *повторно реализовать интерфейс*, указав его имя в списке предков класса наряду с классом-предком, уже реализовавшим этот интерфейс. При этом реализация переопределенных методов базового класса во внимание не принимается:

```
interface IBase
{
   void A();
```

```
}

class Base : IBase
{
   void IBase.A() { ... }        // не используется в Derived
}

class Derived : Base, IBase
{
   public void A() { ... }
}
```

Если класс наследует от класса и интерфейса, которые содержат методы с одинаковыми сигнатурами, унаследованный метод класса воспринимается как реализация интерфейса, например:

```
interface Interface1
{
   void F();
}

class Class1
{
   public void F() { ... }
   public void G() { ... }
}

class Class2 : Class1, Interface1
{
   new public void G() { ... }
}
```

Здесь класс `Class2` наследует от класса `Class1` метод `F`. Интерфейс `Interface1` также содержит метод `F`. Компилятор не выдает ошибку, потому что класс `Class2` содержит метод, подходящий для реализации интерфейса.

Вообще при реализации интерфейса учитывается наличие «подходящих» методов в классе независимо от их происхождения. Это могут быть методы, описанные в текущем или базовом классе, реализующие интерфейс явным или неявным образом.

Стандартные интерфейсы .NET

В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс `IComparable` задает метод сравнения объектов по принципу больше или меньше, что позволяет выполнять их сортировку. Реализация интерфейсов `IEnumerable` и `IEnumerator` дает возможность просматривать содержимое объекта с помощью конструкции `foreach`, а реализация интерфейса `ICloneable` — клонировать объекты.

Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Например, работа с массивами с помощью цикла foreach возможна именно потому, что тип Array реализует интерфейсы IEnumerable и IEnumerator. Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

Сравнение объектов (интерфейс IComparable)

Интерфейс IComparable определен в пространстве имен System. Он содержит всего один метод CompareTo, возвращающий результат сравнения двух объектов — текущего и переданного ему в качестве параметра:

```
interface IComparable
{
    int CompareTo( object obj )
}
```

Метод должен возвращать:

- 0, если текущий объект и параметр равны;
- отрицательное число, если текущий объект меньше параметра;
- положительное число, если текущий объект больше параметра.

Реализуем интерфейс IComparable в знакомом нам классе Monster. В качестве критерия сравнения объектов выберем поле health. В листинге 9.1 приведена программа, сортирующая массив монстров по возрастанию величины, характеризующей их здоровье (элементы класса, не используемые в данной программе, не приводятся).

Листинг 9.1. Пример реализации интерфейса IComparable

```
using System;
namespace ConsoleApplication1
{
    class Monster : IComparable
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }

        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        public int CompareTo( object obj )          // реализация интерфейса
```

продолжение

Листинг 9.1 (*продолжение*)

```
        {
            Monster temp = (Monster) obj;
            if ( this.health > temp.health ) return  1;
            if ( this.health < temp.health ) return -1;
            return 0;
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            const int n = 3;
            Monster[] stado = new Monster[n];

            stado[0] = new Monster( 50, 50, "Вася" );
            stado[1] = new Monster( 80, 80, "Петя" );
            stado[2] = new Monster( 40, 10, "Маша" );

            Array.Sort( stado );          // сортировка стала возможной
            foreach ( Monster elem in stado ) elem.Passport();
        }
    }
}
```

Результат работы программы:

```
Monster Маша     health = 40 ammo = 10
Monster Вася     health = 50 ammo = 50
Monster Петя     health = 80 ammo = 80
```

Если несколько объектов имеют одинаковое значение критерия сортировки, их относительный порядок следования после сортировки не изменится.

Во многих алгоритмах требуется выполнять сортировку объектов по различным критериям. В C# для этого используется интерфейс `IComparer`, который рассмотрен в следующем разделе.

Сортировка по разным критериям (интерфейс IComparer)

Интерфейс `IComparer` определен в пространстве имен `System.Collections`. Он содержит один метод `Compare`, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

```
interface IComparer
{
    int Compare ( object ob1, object ob2 )
}
```

Принцип применения этого интерфейса состоит в том, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий этот интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента (существует несколько перегруженных версий этого метода).

Пример сортировки массива объектов из предыдущего листинга по именам (свойство Name, класс SortByName) и количеству вооружений (свойство Ammo, класс SortByAmmo) приведен в листинге 9.2. Классы параметров сортировки объявлены вложенными, поскольку они требуются только объектам класса Monster.

Листинг 9.2. Сортировка по двум критериям

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
    class Monster
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }

        public int Ammo
        {
            get { return ammo; }
            set
            {
                if (value > 0) ammo = value;
                else           ammo = 0;
            }
        }

        public string Name
        {
            get { return name; }
        }

        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        public class SortByName : IComparer                                    //
        {
            int IComparer.Compare( object ob1, object ob2 )
```

продолжение

Листинг 9.2 (*продолжение*)

```
            {
                Monster m1 = (Monster) ob1;
                Monster m2 = (Monster) ob2;
                return String.Compare( m1.Name, m2.Name );
            }
        }

        public class SortByAmmo : IComparer                                    //
        {
            int IComparer.Compare( object ob1, object ob2 )
            {
                Monster m1 = (Monster) ob1;
                Monster m2 = (Monster) ob2;
                if ( m1.Ammo > m2.Ammo ) return  1;
                if ( m1.Ammo < m2.Ammo ) return -1;
                return 0;
            }
        }
        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            const int n = 3;
            Monster[] stado = new Monster[n];

            stado[0] = new Monster( 50, 50, "Вася" );
            stado[1] = new Monster( 80, 80, "Петя" );
            stado[2] = new Monster( 40, 10, "Маша" );

            Console.WriteLine( "Сортировка по имени:" );
            Array.Sort( stado, new Monster.SortByName() );
            foreach ( Monster elem in stado ) elem.Passport();

            Console.WriteLine( "Сортировка по вооружению:" );
            Array.Sort( stado, new Monster.SortByAmmo() );
            foreach ( Monster elem in stado ) elem.Passport();
        }
    }
}
```

Результат работы программы:

```
Сортировка по имени:
Monster Вася     health = 50 ammo = 50
Monster Маша     health = 40 ammo = 10
Monster Петя     health = 80 ammo = 80
```

```
Сортировка по вооружению:
Monster Маша      health = 40 ammo = 10
Monster Вася      health = 50 ammo = 50
Monster Петя      health = 80 ammo = 80
```

Перегрузка операций отношения

Если класс реализует интерфейс IComparable, его экземпляры можно сравнивать между собой по принципу больше или меньше. Логично разрешить использовать для этого операции отношения, перегрузив их. Операции должны перегружаться парами: < и >, <= и >=, == и !=. Перегрузка операций обычно выполняется путем делегирования, то есть обращения к переопределенным методам CompareTo и Equals.

ПРИМЕЧАНИЕ

Если класс реализует интерфейс IComparable, требуется переопределить метод Equals и связанный с ним метод GetHashCode. Оба метода унаследованы от базового класса object. Пример перегрузки был приведен в разделе «Класс object» (см. с. 183).

В листинге 9.3 операции отношения перегружены для класса Monster. В качестве критерия сравнения объектов по принципу больше или меньше выступает поле health, а при сравнении на равенство реализуется значимая семантика, то есть попарно сравниваются все поля объектов

Листинг 9.3. Перегрузка операций отношения

```
using System;
namespace ConsoleApplication1
{
    class Monster : IComparable
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }

        public override bool Equals( object obj )
        {
            if ( obj == null || GetType() != obj.GetType() ) return false;

            Monster temp = (Monster) obj;
            return health == temp.health &&
                   ammo   == temp.ammo   &&
                   name   == temp.name;
        }

        public override int GetHashCode()
```

продолжение

Листинг 9.3 (*продолжение*)

```
        {
            return name.GetHashCode();
        }

        public static bool operator == ( Monster a, Monster b )
        {
            return a.Equals( b );
        }

//  вариант:
//      public static bool operator == ( Monster a, Monster b )
//      {
//          return ( a.CompareTo( b ) == 0 );
//      }

        public static bool operator != ( Monster a, Monster b )
        {
            return ! a.Equals( b );
        }

//  вариант:
//      public static bool operator != ( Monster a, Monster b )
//      {
//          return ( a.CompareTo( b ) != 0 );
//      }

        public static bool operator < ( Monster a, Monster b )
        {
            return ( a.CompareTo( b ) < 0 );
        }

        public static bool operator > ( Monster a, Monster b )
        {
            return ( a.CompareTo( b ) > 0 );
        }

        public static bool operator <= ( Monster a, Monster b )
        {
            return ( a.CompareTo( b ) <= 0 );
        }

        public static bool operator >= ( Monster a, Monster b )
        {
            return ( a.CompareTo( b ) >= 0 );
        }

        public int CompareTo( object obj )
        {
```

```
            Monster temp = (Monster) obj;
            if ( this.health > temp.health ) return  1;
            if ( this.health < temp.health ) return -1;
            return 0;
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            Monster Вася = new Monster( 70, 80, "Вася" );
            Monster Петя = new Monster( 80, 80, "Петя" );

            if      ( Вася > Петя )  Console.WriteLine( "Вася больше Пети" );
            else if ( Вася == Петя ) Console.WriteLine( "Вася == Петя" );
            else                     Console.WriteLine( "Вася меньше Пети" );
        }
    }
}
```

Результат работы программы не разочаровывает:

```
Вася меньше Пети
```

Клонирование объектов (интерфейс ICloneable)

Клонирование — это создание копии объекта. Копия объекта называется клоном. Как вам известно, при присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект (рис. 9.1, *а*). Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом `MemberwiseClone`, который любой объект наследует от класса `object`. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются (рис. 9.1, *б*). Это называется *поверхностным клонированием*.

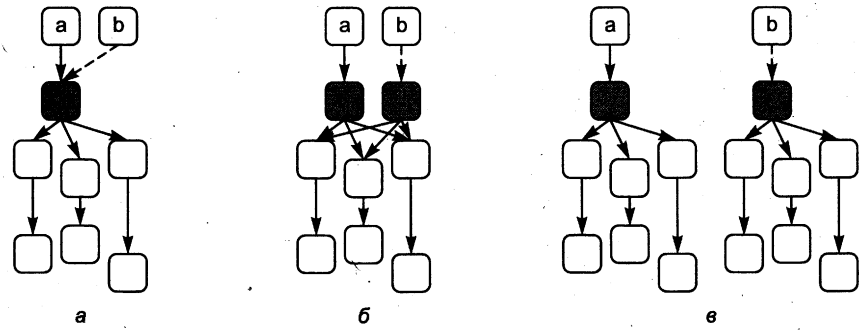


Рис. 9.1. Клонирование объектов

Для создания полностью независимых объектов необходимо *глубокое клонирование*, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей и т. д. (рис. 9.1, *в*). Алгоритм глубокого клонирования весьма сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса `ICloneable` и переопределять его единственный метод `Clone`. В листинге 9.4 приведен пример создания поверхностной копии объекта класса `Monster` с помощью метода `MemberwiseClone`, а также реализован интерфейс `ICloneable`. В демонстрационных целях в имя клона объекта добавлено слово «Клон».

Обратите внимание на то, что метод `MemberwiseClone` можно вызвать только из методов класса. Он не может быть вызван непосредственно, поскольку объявлен в классе `object` как защищенный (`protected`).

Листинг 9.4. Клонирование объектов

```
using System;
namespace ConsoleApplication1
{
    class Monster : ICloneable
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }
        public Monster ShallowClone()                  // поверхностная копия
        {
            return (Monster)this.MemberwiseClone();
        }

        public object Clone()                        // пользовательская копия
        {
            return new Monster( this.health, this.ammo, "Клон " + this.name );
        }

        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
```

```
        {
            Monster Вася = new Monster( 70, 80, "Вася" );
            Monster X = Вася;
            Monster Y = Вася.ShallowClone();
            Monster Z = (Monster)Вася.Clone();
            ...
        }
    }
}
```

Объект X ссылается на ту же область памяти, что и объект Вася. Следовательно, если мы внесем изменения в один из этих объектов, это отразится на другом. Объекты Y и Z, созданные путем клонирования, обладают собственными копиями значений полей и независимы от исходного объекта.

Перебор объектов (интерфейс IEnumerable) и итераторы

Оператор foreach является удобным средством перебора элементов объекта. Массивы и все стандартные коллекции библиотеки .NET позволяют выполнять такой перебор благодаря тому, что в них реализованы интерфейсы IEnumerable и IEnumerator. Для применения оператора foreach к пользовательскому типу данных требуется реализовать в нем эти интерфейсы. Давайте посмотрим, как это делается.

Интерфейс IEnumerable (*перечислимый*) определяет всего один метод — GetEnumerator, возвращающий объект типа IEnumerator (*перечислитель*), который можно использовать для просмотра элементов объекта.

Интерфейс IEnumerator задает три элемента:

- свойство Current, возвращающее текущий элемент объекта;
- метод MoveNext, продвигающий перечислитель на следующий элемент объекта;
- метод Reset, устанавливающий перечислитель в начало просмотра.

Цикл foreach использует эти методы для перебора элементов, из которых состоит объект.

Таким образом, если требуется, чтобы для перебора элементов класса мог применяться цикл foreach, необходимо реализовать четыре метода: GetEnumerator, Current, MoveNext и Reset. Например, если внутренние элементы класса организованы в массив, потребуется описать закрытое поле класса, хранящее текущий индекс в массиве, в методе MoveNext задать изменение этого индекса на 1 с проверкой выхода за границу массива, в методе Current — возврат элемента массива по текущему индексу и т. д.

Это не интересная работа, а выполнять ее приходится часто, поэтому в версию 2.0 были введены средства, облегчающие выполнение перебора в объекте — итераторы.

Итератор представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла foreach выполняется один шаг

итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова yield.

Рассмотрим создание итератора на примере (листинг 9.5). Пусть требуется создать объект, содержащий боевую группу экземпляров типа Monster, неоднократно использованного в примерах этой книги. Для простоты ограничим максимальное количество бойцов в группе десятью.

Листинг 9.5. Класс с итератором

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
    class Monster { ... }
    class Daemon  { ... }
    class Stado : IEnumerable                                          // 1
    {
        private Monster[] mas;
        private int n;

        public Stado()
        {
            mas = new Monster[10];
            n = 0;
        }

        public IEnumerator GetEnumerator()
        {
            for ( int i = 0; i < n; ++i ) yield return mas[i];         // 2
        }

        public void Add( Monster m )
        {
            if ( n >= 10 ) return;
            mas[n] = m;
            ++n;
        }
    }

    class Class1
    {   static void Main()
        {
            Stado s = new Stado();
            s.Add( new Monster() );
            s.Add( new Monster("Вася") );
            s.Add( new Daemon() );

            foreach ( Monster m in s ) m.Passport();
        }
    }
}
```

Все, что требуется сделать в версии 2.0 для поддержки перебора, — указать, что класс реализует интерфейс IEnumerable (оператор 1), и описать итератор (оператор 2). Доступ к нему может быть осуществлен через методы MoveNext и Current интерфейса IEnumerator.

За кодом, приведенным в листинге 9.5, стоит большая внутренняя работа компилятора. На каждом шаге цикла foreach для итератора создается «оболочка» — служебный объект, который запоминает текущее состояние итератора и выполняет все необходимое для доступа к просматриваемым элементам объекта. Иными словами, код, составляющий итератор, не выполняется так, как он выглядит — в виде непрерывной последовательности, а разбит на отдельные итерации, между которыми состояние итератора сохраняется.

В листинге 9.6 приведен пример итератора, перебирающего четыре заданных строки.

Листинг 9.6. Простейший итератор

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
    class Num : IEnumerable
    {
        public IEnumerator GetEnumerator()
        {
            yield return "one";
            yield return "two";
            yield return "three";
            yield return "oops";
        }
    }

    class Class1
    {   static void Main()
        {
            foreach ( string s in new Num() ) Console.WriteLine( s );
        }
    }
}
```

Результат работы программы:

```
one
two
three
oops
```

Следующий пример демонстрирует перебор значений в заданном диапазоне (от 1 до 5):

```
using System;
using System.Collections;
```

```
namespace ConsoleApplication1
{   class Class1
    {
        public static IEnumerable Count( int from, int to )
        {
            from = 1;
            while ( from <= to ) yield return from++;
        }
        static void Main()
        {
            foreach ( int i in Count( 1, 5 ) ) Console.WriteLine( i );
        }
    }
}
```

Преимущество использования итераторов заключается в том, что для одного и того же класса можно задать различный порядок перебора элементов. В листинге 9.7 описаны две дополнительные стратегии перебора элементов класса Stado, введенного в листинге 9.5, — перебор в обратном порядке и выборка только тех объектов, которые являются экземплярами класса Monster (для этого использован метод получения типа объекта GetType, унаследованный от базового класса object).

Листинг 9.7. Реализация нескольких стратегий перебора

```
using System;
using System.Collections;
using MonsterLib;

namespace ConsoleApplication1
{
    class Monster { ... }
    class Daemon  { ... }
    class Stado : IEnumerable
    {
        private Monster[] mas;
        private int n;
        public Stado()
        {
            mas = new Monster[10];
            n = 0;
        }
        public IEnumerator GetEnumerator()
        {
            for ( int i = 0; i < n; ++i ) yield return mas[i];
        }
        public IEnumerable Backwards()                  // в обратном порядке
        {
            for ( int i = n - 1; i >= 0; --i ) yield return mas[i];
        }

        public IEnumerable MonstersOnly()                  // только монстры
```

```
        {
            for ( int i = 0; i < n; ++i )
                if ( mas[i].GetType().Name == "Monster" )
                    yield return mas[i];
        }
        public void Add( Monster m )
        {
            if ( n >= 10 ) return;
            mas[n] = m;
            ++n;
        }
    }

    class Class1
    {   static void Main()
        {
            Stado s = new Stado();
            s.Add( new Monster() );
            s.Add( new Monster("Вася") );
            s.Add( new Daemon() );

            foreach ( Monster i in s )                i.Passport();
            foreach ( Monster i in s.Backwards() )    i.Passport();
            foreach ( Monster i in s.MonstersOnly() ) i.Passport();
        }
    }
}
```

Теперь, когда вы получили представление об итераторах, рассмотрим их более формально.

Блок итератора синтаксически представляет собой обычный блок и может встречаться в теле метода, операции или части get свойства, если соответствующее возвращаемое значение имеет тип IEnumerable или IEnumerator[1].

В теле блока итератора могут встречаться две конструкции:

- yield return формирует значение, выдаваемое на очередной итерации;
- yield break сигнализирует о завершении итерации.

Ключевое слово yield имеет специальное значение для компилятора только в этих конструкциях.

Код блока итератора выполняется не так, как обычные блоки. Компилятор формирует служебный *объект-перечислитель*, при вызове метода MoveNext которого выполняется код блока итератора, выдающий очередное значение с помощью ключевого слова yield. Следующий вызов метода MoveNext объекта-перечислителя возобновляет выполнение блока итератора с момента, на котором он был приостановлен в предыдущий раз.

---

[1] А также тип их параметризованных двойников IEnumerable<T> или IEnumerator<T> из пространства имен System.Collections.Generic, описанного в главе 13.

Структуры

Структура — тип данных, аналогичный классу, но имеющий ряд важных отличий от него:

- структура является *значимым*, а не ссылочным типом данных, то есть экземпляр структуры хранит значения своих элементов, а не ссылки на них, и располагается в стеке, а не в хипе;
- структура не может участвовать в иерархиях наследования, она может только реализовывать интерфейсы;
- в структуре запрещено определять конструктор по умолчанию, поскольку он определен неявно и присваивает всем ее элементам значения по умолчанию (нули соответствующего типа);
- в структуре запрещено определять деструкторы, поскольку это бессмысленно.

ПРИМЕЧАНИЕ

Строго говоря, любой значимый тип C# является структурным.

Отличия от классов обусловливают *область применения структур*: типы данных, имеющие небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками. Накладные расходы на динамическое выделение памяти для небольших объектов могут весьма значительно снизить быстродействие программы, поэтому их эффективнее описывать как структуры, а не как классы.

ПРИМЕЧАНИЕ

С другой стороны, передача структуры в метод по значению требует и дополнительного времени, и дополнительной памяти.

Синтаксис структуры:

```
[ атрибуты ] [ спецификаторы ] struct имя_структуры [ : интерфейсы ]
    тело_структуры [ ; ]
```

Спецификаторы структуры имеют такой же смысл, как и для класса, причем из спецификаторов доступа допускаются только `public`, `internal` и `private` (последний — только для вложенных структур).

Интерфейсы, реализуемые структурой, перечисляются через запятую. *Тело структуры* может состоять из констант, полей, методов, свойств, событий, индексаторов, операций, конструкторов и вложенных типов. Правила их описания и использования аналогичны соответствующим элементам классов, за исключением некоторых отличий, вытекающих из упомянутых ранее:

- поскольку структуры не могут участвовать в иерархиях, для их элементов не могут использоваться спецификаторы `protected` и `protected internal`;
- структуры не могут быть абстрактными (`abstract`), к тому же по умолчанию они бесплодны (`sealed`);

- методы структур не могут быть абстрактными и виртуальными;
- переопределяться (то есть описываться со спецификатором override) могут только методы, унаследованные от базового класса object;
- параметр this интерпретируется как значение, поэтому его можно использовать для ссылок, но не для присваивания;
- при описании структуры нельзя задавать значения полей по умолчанию[1] — это будет сделано в конструкторе по умолчанию, создаваемом автоматически (конструктор присваивает значимым полям структуры нули, а ссылочным — значение null).

В листинге 9.8 приведен пример описания структуры, представляющей комплексное число. Для экономии места из всех операций приведено только описание сложения. Обратите внимание на перегруженный метод ToString: он позволяет выводить экземпляры структуры на консоль, поскольку неявно вызывается в методе Console.WriteLine. Использованные в методе спецификаторы формата описаны в приложении.

Листинг 9.8. Пример структуры

```
using System;
namespace ConsoleApplication1
{
    struct Complex
    {
        public double re, im;

        public Complex( double re_, double im_ )
        {
            re = re_; im = im_;        // можно использовать this.re, this.im
        }

        public static Complex operator + ( Complex a, Complex b )
        {
            return new Complex( a.re + b.re, a.im + b.im );
        }

        public override string ToString()
        {
            return ( string.Format( "({0,2:0.##};{1,2:0.##})", re, im ) );
        }
    }

    class Class1
    {   static void Main()
        {
            Complex a = new Complex( 1.2345, 5.6 );
```

продолжение

[1] К статическим полям это ограничение не относится.

Листинг 9.8 (*продолжение*)

```
            Console.WriteLine( "a = " + a );

            Complex b;
            b.re = 10; b.im = 1;
            Console.WriteLine( "b = " + b );

            Complex c = new Complex();
            Console.WriteLine( "c = " + c );

            c = a + b;
            Console.WriteLine( "c = " + c );
        }
    }
}
```

Результат работы программы:

```
a = (1,23;5,6)
b = (10; 1)
c = ( 0; 0)
c = (11,23;6,6)
```

При выводе экземпляра структуры на консоль выполняется *упаковка*, то есть неявное преобразование в ссылочный тип. Упаковка (это понятие было введено в разделе «Упаковка и распаковка», см. с. 36) применяется и в других случаях, когда структурный тип используется там, где ожидается ссылочный, например, при преобразовании экземпляра структуры к типу реализуемого ею интерфейса. При обратном преобразовании — из ссылочного типа в структурный — выполняется *распаковка*.

Присваивание структур имеет, что естественно, значимую семантику, то есть при присваивании создается копия значений полей. То же самое происходит и при передаче структур в качестве параметров по значению. Для экономии ресурсов ничто не мешает передавать структуры в методы по ссылке с помощью ключевых слов `ref` или `out`.

Особенно значительный выигрыш в эффективности можно получить, используя массивы структур вместо массивов классов. Например, для массива из 100 экземпляров класса создается 101 объект, а для массива структур — один объект. Пример работы с массивом структур, описанных в предыдущем листинге:

```
Complex [] mas = new Complex[4];

for ( int i = 0; i < 4; ++i )
{
    mas[i].re = i;
    mas[i].im = 2 * i;
}

foreach ( Complex elem in mas ) Console.WriteLine( elem );
```

Если поместить этот фрагмент вместо тела метода Main в листинге 9.5, получим следующий результат:

```
( 0; 0)
( 1; 2)
( 2; 4)
( 3; 6)
```

Перечисления

При написании программ часто возникает потребность определить несколько связанных между собой именованных констант, при этом их конкретные значения могут быть не важны. Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант, например:

```
enum Menu { Read, Write, Append, Exit }
enum Радуга { Красный, Оранжевый, Желтый, Зеленый, Синий, Фиолетовый }
```

Для каждой константы задается ее символическое имя. По умолчанию константам присваиваются последовательные значения типа int, начиная с 0, но можно задать и собственные значения, например:

```
enum Nums { two = 2, three, four, ten = 10, eleven, fifty = ten + 40 };
```

Константам three и four присваиваются значения 3 и 4, константе eleven — 11. Имена перечисляемых констант внутри каждого перечисления должны быть уникальными, а значения могут совпадать.

Преимущество перечисления перед описанием именованных констант состоит в том, что связанные константы нагляднее; кроме того, компилятор выполняет проверку типов, а интегрированная среда разработки подсказывает возможные значения констант, выводя их список.

Синтаксис перечисления:

```
[ атрибуты ] [ спецификаторы ] enum имя_перечисления [ : базовый_тип ]
    тело_перечисления [ ; ]
```

Спецификаторы перечисления имеют такой же смысл, как и для класса, причем допускаются только спецификаторы new, public, protected, internal и private.

Базовый тип — это тип элементов, из которых построено перечисление. По умолчанию используется тип int, но можно задать тип и явным образом, выбрав его среди целочисленных типов (кроме char), а именно: byte, sbyte, short, ushort, int, uint, long и ulong. Необходимость в этом возникает, когда значения констант невозможно или неудобно представлять с помощью типа int.

Тело перечисления состоит из имен констант, каждой из которых может быть присвоено значение. Если значение не указано, оно вычисляется прибавлением единицы к значению предыдущей константы. Константы по умолчанию имеют спецификатор доступа public.

Перечисления часто используются как вложенные типы, идентифицируя значения из какого-либо ограниченного набора. Пример такого перечисления приведен в листинге 9.9.

Листинг 9.9. Пример перечисления

```
using System;
namespace ConsoleApplication1
{
    struct Боец
    {
        public enum Воинское_Звание
        {
            Рядовой, Сержант, Майор, Генерал
        }

        public string Фамилия;
        public Воинское_Звание Звание;
    }
    class Class1
    {   static void Main()
        {
            Боец x;
            x.Фамилия = "Иванов";
            x.Звание = Боец.Воинское_Звание.Сержант;

            Console.WriteLine( x.Звание + " " + x.Фамилия );
        }
    }
}
```

Результат работы программы:

```
Сержант Иванов
```

Перечисления удобно использовать для представления битовых флагов, например:

```
enum Flags : byte
{
    b0, b1, b2, b3 = 0x04, b4 = 0x08, b5 = 0x10, b6 = 0x20, b7 = 0x40
}
```

Операции с перечислениями

С переменными перечисляемого типа можно выполнять арифметические операции (+, -, ++, --), логические поразрядные операции (^, &, |, ~), сравнивать их с помощью операций отношения (<, <=, >, >=, ==, !=) и получать размер в байтах (`sizeof`).

При использовании переменных перечисляемого типа в целочисленных выражениях и операциях присваивания требуется явное *преобразование типа*. Переменной перечисляемого типа можно присвоить любое значение, представимое с помощью

базового типа, то есть не только одно из значений, входящих в тело перечисления. Присваиваемое значение становится новым элементом перечисления.

Пример:

```
Flags a = Flags.b2 | Flags.b4;
Console.WriteLine( "a = {0}   {0,2:X}", a );
++a;

Console.WriteLine( "a = {0}   {0,2:X}", a );
int x = (int) a;
Console.WriteLine( "x = {0}   {0,2:X}", x );

Flags b = (Flags) 65;
Console.WriteLine( "b = {0}   {0,2:X}", b );
```

Результат работы этого фрагмента программы ({0,2:X} обозначает шестнадцатеричный формат вывода):

```
a = 10    0A
a = 11    0B
x = 11     B
b = 65    41
```

Другой пример использования операций с перечислениями приведен в листинге 9.10.

Листинг 9.10. Операции с перечислениями

```
using System;
namespace ConsoleApplication1
{
    struct Боец
    {
        public enum Воинское_Звание
        {
            Рядовой, Сержант, Лейтенант, Майор, Полковник, Генерал
        }

        public string Фамилия;
        public Воинское_Звание Звание;
    }

    class Class1
    {   static void Main()
        {
            Боец x;
            x.Фамилия = "Иванов";
            x.Звание  = Боец.Воинское_Звание.Сержант;

            for ( int i = 1976; i < 2006; i += 5 )
            {
                if ( x.Звание < Боец.Воинское_Звание.Генерал ) ++x.Звание;
```

продолжение

Листинг 9.10 (*продолжение*)

```
                Console.WriteLine( "Год: {0} {1} {2}",
                                   i, x.Звание, x.Фамилия );
            }
        }
    }
}
```

Результат работы программы:

```
Год: 1976 Лейтенант Иванов
Год: 1981 Майор Иванов
Год: 1986 Полковник Иванов
Год: 1991 Генерал Иванов
Год: 1996 Генерал Иванов
Год: 2001 Генерал Иванов
```

Базовый класс System.Enum

Все перечисления в C# являются потомками базового класса System.Enum, который снабжает их некоторыми полезными методами.

Статический метод GetName позволяет получить символическое имя константы по ее номеру, например:

```
Console.WriteLine( Enum.GetName(typeof( Flags ), 8) );                    // b4
Console.WriteLine( Enum.GetName(typeof( Боец.Воинское_Звание ), 1) );  // Сержант
```

ПРИМЕЧАНИЕ

Операция typeof возвращает тип своего аргумента (см. раздел «Рефлексия» в главе 12).

Статические методы GetNames и GetValues формируют, соответственно, массивы имен и значений констант, составляющих перечисление, например:

```
Array names = Enum.GetNames( typeof(Flags) );
Console.WriteLine( "Количество элементов в перечислении: " + names.Length );
foreach ( string elem in names ) Console.Write( "  " + elem );

Array values = Enum.GetValues( typeof(Flags) );
foreach ( Flags elem in values ) Console.Write( "  " + (byte) elem );
```

Статический метод IsDefined возвращает значение true, если константа с заданным символическим именем описана в указанном перечислении, и false в противном случае, например:

```
if ( Enum.IsDefined( typeof( Flags ), "b5" ) )
     Console.WriteLine( "Константа с именем b5 существует" );
else Console.WriteLine( "Константа с именем b5 не существует" );
```

Статический метод GetUnderlyingType возвращает имя базового типа, на котором построено перечисление. Например, для перечисления Flags будет получено System.Byte:

```
Console.WriteLine( Enum.GetUnderlyingType( typeof(Flags) ) );
```

Рекомендации по программированию

Интерфейсы чаще всего используются для задания общих свойств объектов различных иерархий. Основная идея интерфейса состоит в том, что к объектам классов, реализующих интерфейс, можно обращаться одинаковым образом, при этом каждый класс может определять элементы интерфейса по-своему.

Если некий набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии.

В C# поддерживается одиночное наследование для классов и множественное — для интерфейсов. Это позволяет придать производному классу свойства нескольких базовых интерфейсов. Класс должен определять все методы всех интерфейсов, которые имеются в списке его предков.

В библиотеке .NET определено большое количество стандартных интерфейсов. Реализация стандартных интерфейсов в собственных классах позволяет использовать для объектов этих классов стандартные средства языка и библиотеки.

Например, для обеспечения возможности сортировки объектов стандартными методами следует реализовать в соответствующем классе интерфейсы `IComparable` или `IComparer`. Реализация интерфейсов `IEnumerable` и `IEnumerator` дает возможность просматривать содержимое объекта с помощью конструкции `foreach`, а реализация интерфейса `ICloneable` — клонировать объекты.

Использование *итераторов* упрощает организацию перебора элементов и позволяет задать для одного и того же класса различные стратегии перебора.

Область применения *структур* — типы данных, имеющие небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками. Накладные расходы на динамическое выделение памяти для экземплятров небольших классов могут весьма значительно снизить быстродействие программы, поэтому их эффективнее описывать как структуры.

Преимущество использования *перечислений* для описания связанных между собой значений состоит в том, что это более наглядно и инкапсулировано, чем россыпь именованных констант. Кроме того, компилятор выполняет проверку типов, а интегрированная среда разработки подсказывает возможные значения констант, выводя их список.

Глава 10
Делегаты, события и потоки выполнения

В этой главе рассматриваются делегаты и события — два взаимосвязанных средства языка C#, позволяющие организовать эффективное взаимодействие объектов. Во второй части главы приводятся начальные сведения о разработке многопоточных приложений.

Делегаты

Делегат — это вид класса, предназначенный для хранения ссылок на методы. Делегат, как и любой другой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод. Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка. Рассмотрим сначала второй случай.

Описание делегатов

Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

```
[ атрибуты ] [ спецификаторы ] delegate тип имя_делегата ( [ параметры ] )
```

Спецификаторы делегата имеют тот же смысл, что и для класса, причем допускаются только спецификаторы `new`, `public`, `protected`, `internal` и `private`.

Тип описывает возвращаемое значение методов, вызываемых с помощью делегата, а необязательными *параметрами* делегата являются параметры этих методов. Делегат может хранить ссылки на несколько методов и вызывать их поочередно; естественно, что сигнатуры всех методов должны совпадать.

Пример описания делегата:

```
public delegate void D ( int i );
```

Здесь описан тип делегата, который может хранить ссылки на методы, возвращающие void и принимающие один параметр целого типа.

ПРИМЕЧАНИЕ

Делегат, как и всякий класс, представляет собой тип данных. Его базовым классом является класс System.Delegate, снабжающий своего «отпрыска» некоторыми полезными элементами, которые мы рассмотрим позже. Наследовать от делегата нельзя, да и нет смысла.

Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса.

Использование делегатов

Для того чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове экземпляра делегата вызываются все заданные в нем методы.

Делегаты применяются в основном для следующих целей:

- получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
- обеспечения связи между объектами по типу «источник — наблюдатель»;
- создания универсальных методов, в которые можно передавать другие методы;
- поддержки механизма обратных вызовов.

Все эти варианты подробно обсуждаются далее. Рассмотрим сначала пример реализации первой из этих целей. В листинге 10.1 объявляется делегат, с помощью которого один и тот же оператор используется для вызова двух разных методов (COOl и Hack).

Листинг 10.1. Простейшее использование делегата

```
using System;
namespace ConsoleApplication1
{
    delegate void Del ( ref string s );                    // объявление делегата

    class Class1
    {
        public static void COOl ( ref string s )                    // метод 1
        {
            string temp = "";
            for ( int i = 0; i < s.Length; ++i )
            {
```

продолжение

Листинг 10.1 (*продолжение*)

```
            if      ( s[i] == 'o' || s[i] == '0') temp += '0';
            else if ( s[i] == 'l' )               temp += '1';
            else                                  temp += s[i];
        }
        s = temp;
    }

    public static void Hack ( ref string s )                    // метод 2
    {
        string temp = "";
        for ( int i = 0; i < s.Length; ++i )
            if ( i / 2 * 2 == i ) temp += char.ToUpper( s[i] );
            else                  temp += s[i];

        s = temp;
    }

    static void Main()
    {
        string s = "cool hackers";
        Del d;                                          // экземпляр делегата

        for ( int i = 0; i < 2; ++i )
        {
            d = new Del( COOl );                // инициализация методом 1
            if ( i == 1 ) d = new Del(Hack);    // инициализация методом 2

            d( ref s );     // использование делегата для вызова методов
            Console.WriteLine( s );
        }
    }
  }
}
```

Результат работы программы:

```
cOOl hackers
COOl hAcKeRs
```

Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит *ссылки на несколько методов*, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

Добавление метода в список выполняется либо с помощью метода `Combine`, унаследованного от класса `System.Delegate`, либо, что удобнее, с помощью перегруженной операции сложения. Вот как выглядит измененный метод `Main` из предыдущего листинга, в котором одним вызовом делегата выполняется преобразование исходной строки сразу двумя методами:

```
static void Main()
{
        string s = "cool hackers";
        Del d = new Del( COOl );
        d += new Del( Hack );              // добавление метода в делегат

        d( ref s );
        Console.WriteLine( s );            // результат: COOl hAcKeRs
}
```

При вызове последовательности методов с помощью делегата необходимо учитывать следующее:

- сигнатура методов должна в точности соответствовать делегату;
- методы могут быть как статическими, так и обычными методами класса;
- каждому методу в списке передается один и тот же набор параметров;
- если параметр передается по ссылке, изменения параметра в одном методе отразятся на его значении при вызове следующего метода;
- если параметр передается с ключевым словом `out` или метод возвращает значение, результатом выполнения делегата является значение, сформированное последним из методов списка (в связи с этим рекомендуется формировать списки только из делегатов, имеющих возвращаемое значение типа `void`);
- если в процессе работы метода возникло исключение, не обработанное в том же методе, последующие методы в списке не выполняются, а происходит поиск обработчиков в объемлющих делегат блоках;
- попытка вызвать делегат, в списке которого нет ни одного метода, вызывает генерацию исключения `System.NullReferenceException`.

Паттерн «наблюдатель»

Рассмотрим применение делегатов для обеспечения связи между объектами по типу «источник — наблюдатель». В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов. При этом желательно избежать жесткой связанности классов, так как это часто негативно сказывается на возможности многократного использования кода.

Для обеспечения гибкой, динамической связи между объектами во время выполнения программы применяется следующая стратегия. Объект, называемый *источником*, при изменении своего состояния, которое может представлять интерес для других объектов, посылает им уведомления. Эти объекты называются *наблюдателями*. Получив уведомление, наблюдатель опрашивает источник, чтобы синхронизировать с ним свое состояние.

Примером такой стратегии может служить связь объекта с различными его представлениями, например, связь электронной таблицы с созданными на ее основе диаграммами.

Программисты часто используют одну и ту же схему организации и взаимодействия объектов в разных контекстах. За такими схемами закрепилось название *паттерны*, или *шаблоны проектирования*. Описанная стратегия известна под названием *паттерн «наблюдатель»*.

Наблюдатель (observer) определяет между объектами зависимость типа «один ко многим», так что при изменении состоянии одного объекта все зависящие от него объекты получают извещение и автоматически обновляются. Рассмотрим пример (листинг 10.2), в котором демонстрируется схема оповещения источником трех наблюдателей. Гипотетическое изменение состояния объекта моделируется сообщением «OOPS!». Один из методов в демонстрационных целях сделан статическим.

Листинг 10.2. Оповещение наблюдателей с помощью делегата

```
using System;
namespace ConsoleApplication1
{
    public delegate void Del( object o );              // объявление делегата

    class Subj                                          // класс-источник
    {
        Del dels;                              // объявление экземпляра делегата

        public void Register( Del d )                  // регистрация делегата
        {
            dels += d;
        }

        public void OOPS()                              // что-то произошло
        {
            Console.WriteLine( "OOPS!" );
            if ( dels != null ) dels( this );      // оповещение наблюдателей
        }
    }

    class ObsA                                          // класс-наблюдатель
    {
        public void Do( object o )             // реакция на событие источника
        {
            Console.WriteLine( "Вижу, что OOPS!" );
        }
    }

    class ObsB                                          // класс-наблюдатель
    {
        public static void See( object o )   // реакция на событие источника
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
```

```
    }

    class Class1
    {
        static void Main()
        {
            Subj s  = new Subj();                  //  объект класса-источника

            ObsA o1 = new ObsA();                  //                  объекты
            ObsA o2 = new ObsA();                  //    класса-наблюдателя

            s.Register( new Del( o1.Do ) );        //      регистрация методов
            s.Register( new Del( o2.Do ) );        // наблюдателей в источнике
            s.Register( new Del( ObsB.See ) );     //  ( экземпляры делегата )

            s.OOPS();                              //    инициирование события
        }
    }
}
```

В источнике объявляется экземпляр делегата, в этот экземпляр заносятся методы тех объектов, которые хотят получать уведомление об изменении состояния источника. Этот процесс называется *регистрацией делегатов*. При регистрации имя метода добавляется к списку. Обратите внимание: для статического метода указывается имя класса, а для обычного метода — имя объекта. При наступлении «часа X» все зарегистрированные методы поочередно вызываются через делегат.

Результат работы программы:

```
OOPS!
Вижу, что OOPS!
Вижу, что OOPS!
Я тоже вижу, что OOPS!
```

Для обеспечения обратной связи между наблюдателем и источником делегат объявлен с параметром типа `object`, через который в вызываемый метод передается ссылка на вызывающий объект. Следовательно, в вызываемом методе можно получать информацию о состоянии вызывающего объекта и посылать ему сообщения (то есть вызывать методы этого объекта).

Связь «источник — наблюдатель» устанавливается во время выполнения программы для каждого объекта по отдельности. Если наблюдатель больше не хочет получать уведомления от источника, можно удалить соответствующий метод из списка делегата с помощью метода `Remove` или перегруженной операции вычитания, например:

```
public void UnRegister( Del d )                    // удаление делегата
{
    dels -= d;
}
```

Операции

Делегаты можно *сравнивать на равенство и неравенство*. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке. Сравнивать можно даже делегаты различных типов при условии, что они имеют один и тот же тип возвращаемого значения и одинаковые списки параметров.

С делегатами одного типа можно *выполнять операции* простого и сложного *присваивания*, например:

```
Del d1 = new Del( o1.Do );      // o1.Do
Del d2 = new Del( o2.Do );      // o2.Do
Del d3 = d1 + d2;               // o1.Do и o2.Do
d3 += d1;                       // o1.Do, o2.Do и o1.Do
d3 -= d2;                       // o1.Do и o1.Do
```

Эти операции могут понадобиться, например, в том случае, если в разных обстоятельствах требуется вызывать разные наборы и комбинации наборов методов.

Делегат, как и строка `string`, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

Передача делегатов в методы

Поскольку делегат является классом, его можно передавать в методы в качестве параметра. Таким образом обеспечивается *функциональная параметризация*: в метод можно передавать не только различные данные, но и различные функции их обработки. Функциональная параметризация применяется для создания универсальных методов и обеспечения возможности обратного вызова.

В качестве простейшего примера *универсального метода* можно привести метод вывода таблицы значений функции, в который передается диапазон значений аргумента, шаг его изменения и вид вычисляемой функции. Этот пример приводится далее.

Обратный вызов (callback) представляет собой вызов функции, передаваемой в другую функцию в качестве параметра. Рассмотрим рис. 10.1. Допустим, в библиотеке описана функция `A`, параметром которой является имя другой функции.

В вызывающем коде описывается функция с требуемой сигнатурой (`B`) и передается в функцию `A`. Выполнение функции `A` приводит к вызову `B`, то есть управление передается из библиотечной функции обратно в вызывающий код.

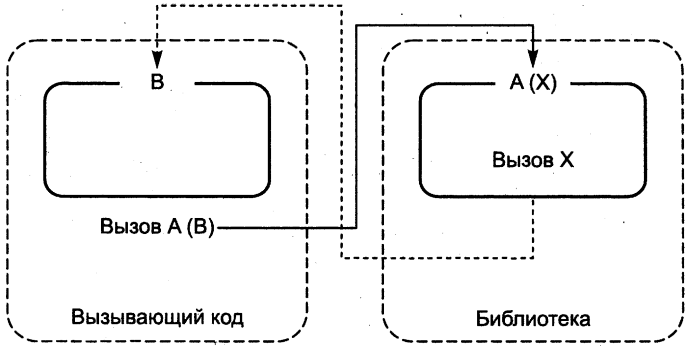


Рис. 10.1. Механизм обратного вызова

Механизм обратного вызова широко используется в программировании. Например, он реализуется во многих стандартных функциях Windows.

Пример передачи делегата в качестве параметра приведен в листинге 10.3. Программа выводит таблицу значений функции на заданном интервале с шагом, равным единице.

Листинг 10.3. Передача делегата через список параметров

```
using System;
namespace ConsoleApplication1
{
    public delegate double Fun( double x );          // объявление делегата

    class Class1
    {
        public static void Table( Fun F, double x, double b )
        {
            Console.WriteLine( " ----- X ----- Y -----" );
            while (x <= b)
            {
               Console.WriteLine( "| {0,8:0.000} | {1,8:0.000} |", x, F(x));
               x += 1;
            }
            Console.WriteLine( " ---------------------" );
        }

        public static double Simple( double x )
        {
            return 1;
        }

        static void Main()
        {
            Console.WriteLine( " Таблица функции Sin " );
```

продолжение

Листинг 10.3 (*продолжение*)

```
            Table( new Fun( Math.Sin ), -2, 2 );

            Console.WriteLine( " Таблица функции Simple " );
            Table( new Fun( Simple ), 0, 3 );
        }
    }
}
```

Результат работы программы:

```
Таблица функции Sin
 ----- X ----- Y -----
|   -2,000 |   -0,909 |
|   -1,000 |   -0,841 |
|    0,000 |    0,000 |
|    1,000 |    0,841 |
|    2,000 |    0,909 |
 --------------------
 Таблица функции Simple
 ----- X ----- Y -----
|    0,000 |    1,000 |
|    1,000 |    1,000 |
|    2,000 |    1,000 |
|    3,000 |    1,000 |
 --------------------
```

В среде Visual Studio 2005, использующей версию 2.0 языка C#, можно применять упрощенный синтаксис для делегатов. Первое упрощение заключается в том, что в большинстве случаев явным образом создавать экземпляр делегата не требуется, поскольку он создается автоматически по контексту. Второе упрощение заключается в возможности создания так называемых *анонимных методов* — фрагментов кода, описываемых непосредственно в том месте, где используется делегат. В листинге 10.4 использованы оба упрощения для реализации тех же действий, что и листинге 10.3.

Листинг 10.4. Передача делегата через список параметров (версия 2.0)

```
using System;
namespace ConsoleApplication1
{
    public delegate double Fun( double x );          // объявление делегата

    class Class1
    {
        public static void Table( Fun F, double x, double b )
        {
            Console.WriteLine( " ----- X ----- Y -----" );
            while ( x <= b )
```

```
            {
                Console.WriteLine( "| {0,8:0.000} | {1,8:0.000} |", x, F(x));
                x += 1;
            }
            Console.WriteLine( " --------------------" );
        }

        static void Main()
        {
            Console.WriteLine( " Таблица функции Sin " );
            Table( Math.Sin, -2, 2 );                          // упрощение 1

            Console.WriteLine( " Таблица функции Simple " );
            Table( delegate (double x ){ return 1; }, 0, 3 ); // упрощение 2
        }
    }
}
```

В первом случае экземпляр делегата, соответствующего функции Sin, создается автоматически[1]. Чтобы это могло произойти, список параметров и тип возвращаемого значения функции должны быть совместимы с делегатом. Во втором случае не требуется оформлять простой фрагмент кода в виде отдельной функции Simple, как это было сделано в предыдущем листинге, — код функции оформляется как анонимный метод и встраивается прямо в место передачи.

Альтернативой использованию делегатов в качестве параметров являются виртуальные методы. Универсальный метод вывода таблицы значений функции можно реализовать с помощью абстрактного базового класса, содержащего два метода: метод вывода таблицы и абстрактный метод, задающий вид вычисляемой функции. Для вывода таблицы конкретной функции необходимо создать производный класс, переопределяющий этот абстрактный метод. Реализация метода вывода таблицы с помощью наследования и виртуальных методов приведена в листинге 10.5.

Листинг 10.5. Альтернатива параметрам-делегатам

```
using System;
namespace ConsoleApplication1
{
    abstract class TableFun
    {
        public abstract double F( double x );

        public void Table( double x, double b )
        {
            Console.WriteLine( " ----- X ----- Y -----" );
            while ( x <= b )
```

продолжение

[1] В результате в 2005 году язык C# в этой части вплотную приблизился к синтаксису старого доброго Паскаля, в котором передача функций в качестве параметров была реализована еще в 1992 году, если не раньше.

Листинг 10.5 (*продолжение*)

```
            {
                Console.WriteLine( "| {0,8:0.000} | {1,8:0.000} |", x, F(x));
                x += 1;
            }
            Console.WriteLine( " ---------------------" );
        }
    }

    class SimpleFun : TableFun
    {
        public override double F( double x )
        {
            return 1;
        }
    }

    class SinFun : TableFun
    {
        public override double F( double x )
        {
            return Math.Sin(x);
        }
    }

    class Class1
    {   static void Main()
        {
            TableFun a = new SinFun();
            Console.WriteLine( " Таблица функции Sin " );
            a.Table( -2, 2 );

            a = new SimpleFun();
            Console.WriteLine( " Таблица функции Simple " );
            a.Table( 0, 3 );
        }
    }
}
```

Результат работы этой программы такой же, как и предыдущей, но, на мой взгляд, в данном случае применение делегатов предпочтительнее.

Обработка исключений при вызове делегатов

Ранее говорилось о том, что если в одном из методов списка делегата генерируется исключение, следующие методы не вызываются. Этого можно избежать, если обеспечить явный перебор всех методов в проверяемом блоке и обрабатывать возникающие исключения. Все методы, заданные в экземпляре делегата, можно

получить с помощью унаследованного метода GetInvocationList. Этот прием иллюстрирует листинг 10.6, представляющий собой измененный вариант листинга 10.1.

Листинг 10.6. Перехват исключений при вызове делегата

```
using System;
namespace ConsoleApplication1
{
    delegate void Del ( ref string s );

    class Class1
    {
        public static void COO1 ( ref string s )
        {
            Console.WriteLine( "вызван метод COO1" );
            string temp = "";
            for ( int i = 0; i < s.Length; ++i )
            {
                if      ( s[i] == 'o' || s[i] == '0') temp += '0';
                else if ( s[i] == '1' )               temp += '1';
                else                                  temp += s[i];
            }
            s = temp;
        }

        public static void Hack ( ref string s )
        {
            Console.WriteLine( "вызван метод Hack" );
            string temp = "";
            for ( int i = 0; i < s.Length; ++i )
                if ( i / 2 * 2 == i ) temp += char.ToUpper( s[i] );
                else                  temp += s[i];

            s = temp;
        }

        public static void BadHack ( ref string s )
        {
            Console.WriteLine( "вызван метод BadHack" );
            throw new Exception();                          // имитация ошибки
        }

        static void Main()
        {
            string s = "cool hackers";
            Del d = new Del( COO1 );          // создание экземпляра делегата
            d += new Del( BadHack );          // дополнение списка методов
            d += new Del( Hack );             // дополнение списка методов
```

продолжение

Листинг 10.6 (*продолжение*)

```
            foreach ( Del fun in d.GetInvocationList() )
            {
                try
                {
                    fun( ref s );          // вызов каждого метода из списка
                }
                catch ( Exception e )
                {
                    Console.WriteLine( e.Message );
                    Console.WriteLine( "Exception in method " +
                                       fun.Method.Name);
                }
            }
            Console.WriteLine( "результат - " + s );
        }
    }
}
```

Результат работы программы:

```
вызван метод C001
вызван метод BadHack
Exception of type System.Exception was thrown.
Exception in method BadHack
вызван метод Hack
результат - C001 hAcKeRs
```

В этой программе помимо метода базового класса `GetInvocationList` использовано свойство `Method`. Это свойство возвращает результат типа `MethodInfo`. Класс `MethodInfo` содержит множество свойств и методов, позволяющих получить полную информацию о методе, например его спецификаторы доступа, имя и тип возвращаемого значения. Мы рассмотрим этот интересный класс в разделе «Рефлексия» главы 12.

События

Событие — это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния. При этом для объектов, являющихся наблюдателями события, активизируются методы-обработчики этого события. Обработчики должны быть зарегистрированы в объекте-источнике события. Таким образом, механизм событий формализует на языковом уровне паттерн «наблюдатель», который рассматривался в предыдущем разделе.

Механизм событий можно также описать с помощью модели «публикация — подписка»: один класс, являющийся *отправителем* (sender) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся *получателями* (receivers) сообщения, подписываются на получение этих событий.

События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому *создание события* в классе состоит из следующих частей:

- описание делегата, задающего сигнатуру обработчиков событий;
- описание события;
- описание метода (методов), инициирующих событие.

Синтаксис события похож на синтаксис делегата[1]:

```
[ атрибуты ] [ спецификаторы ] event тип имя_события
```

Для событий применяются *спецификаторы* new, public, protected, internal, private, static, virtual, sealed, override, abstract и extern, которые изучались при рассмотрении методов классов. Например, так же как и методы, событие может быть статическим (static), тогда оно связано с классом в целом, или обычным — в этом случае оно связано с экземпляром класса.

Тип события — это тип делегата, на котором основано событие.

Пример описания делегата и соответствующего ему события:

```
public delegate void Del( object o );                    // объявление делегата
class A
{
    public event Del Oops;                               // объявление события
    ...
}
```

Обработка событий выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответствует типу делегата. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.

Как видите, это в точности тот же самый механизм, который рассматривался в предыдущем разделе. Единственное отличие состоит в том, что при использовании событий не требуется описывать метод, регистрирующий обработчики, поскольку события поддерживают операции += и -=, добавляющие обработчик в список и удаляющие его из списка.

ПРИМЕЧАНИЕ

Событие — это удобная абстракция для программиста. На самом деле оно состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.

В листинге 10.7 приведен код из листинга 10.2, переработанный с использованием событий.

[1] Приводится упрощенный вариант. В общем случае имеется возможность задавать несколько имен событий, инициализаторы и методы добавления и удаления событий.

Листинг 10.7. Оповещение наблюдателей с помощью событий

```
using System;
namespace ConsoleApplication1
{
    public delegate void Del();                          // объявление делегата

    class Subj                                                // класс-источник
    {
        public event Del Oops;                           // объявление события

        public void CryOops()                  // метод, инициирующий событие
        {
            Console.WriteLine( "OOPS!" );
            if ( Oops != null ) Oops();
        }
    }

    class ObsA                                               // класс-наблюдатель
    {
        public void Do();                      // реакция на событие источника
        {
            Console.WriteLine( "Вижу, что OOPS!" );
        }
    }

    class ObsB                                               // класс-наблюдатель
    {
        public static void See()               // реакция на событие источника
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
    }

    class Class1
    {
        static void Main()
        {
            Subj s  = new Subj();              //    объект класса-источника

            ObsA o1 = new ObsA();              //                    объекты
            ObsA o2 = new ObsA();              //-       класса-наблюдателя

            s.Oops += new Del( o1.Do );        //                 добавление
            s.Oops += new Del( o2.Do );        //               обработчиков
            s.Oops += new Del( ObsB.See );     //                  к событию

            s.CryOops();                       //      инициирование события
        }
    }
}
```

Внешний код может работать с событиями единственным образом: добавлять обработчики в список или удалять их, поскольку вне класса могут использоваться только операции += и -=. Тип результата этих операций — void, в отличие от операций сложного присваивания для арифметических типов. Иного способа доступа к списку обработчиков нет.

Внутри класса, в котором описано событие, с ним можно обращаться, как с обычным полем, имеющим тип делегата: использовать операции отношения, присваивания и т. д. Значение события по умолчанию — null. Например, в методе CryOops выполняется проверка на null для того, чтобы избежать генерации исключения System.NullReferenceException.

В библиотеке .NET описано огромное количество стандартных делегатов, предназначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам:

- имя делегата заканчивается суффиксом EventHandler;
- делегат получает два параметра:
  - первый параметр задает источник события и имеет тип object;
  - второй параметр задает аргументы события и имеет тип EventArgs или производный от него.

Если обработчикам события требуется специфическая информация о событии, то для этого создают класс, производный от стандартного класса EventArgs, и добавляют в него необходимую информацию. Если делегат не использует такую информацию, можно не описывать делегата и собственный тип аргументов, а обойтись стандартным классом делегата System.EventHandler.

Имя обработчика события принято составлять из префикса On и имени события. В листинге 10.8 приведен пример из листинга 10.7, оформленный в соответствии со стандартными соглашениями .NET. Найдите восемь отличий!

Листинг 10.8. Использование стандартного делегата EventHandler

```
using System;
namespace ConsoleApplication1
{
    class Subj
    {
        public event EventHandler Oops;

        public void CryOops()
        {
            Console.WriteLine( "OOPS!" );
            if ( Oops != null ) Oops( this, null );
        }
    }

    class ObsA
    {
```

продолжение

Листинг 10.8 (*продолжение*)

```
        public void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Вижу, что OOPS!" );
        }
    }

    class ObsB
    {
        public static void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
    }

    class Class1
    {   static void Main()
        {
            Subj s  = new Subj();

            ObsA o1 = new ObsA();
            ObsA o2 = new ObsA();

            s.Oops += new EventHandler( o1.OnOops );
            s.Oops += new EventHandler( o2.OnOops );
            s.Oops += new EventHandler( ObsB.OnOops );

            s.CryOops();
        }
    }
}
```

Те, кто работает с C# версии 2.0, могут упростить эту программу, используя новую возможность неявного создания делегатов при регистрации обработчиков событий. Соответствующий вариант приведен в листинге 10.9. В демонстрационных целях в код добавлен новый *анонимный обработчик* — еще один механизм, появившийся в новой версии языка.

Листинг 10.9. Использование делегатов и анонимных методов (версия 2.0)

```
using System;
namespace ConsoleApplication1
{
    class Subj
    {
        public event EventHandler Oops;

        public void CryOops()
        {
            Console.WriteLine( "OOPS!" );
```

```
            if ( Oops != null ) Oops( this, null );
        }
    }

    class ObsA
    {
        public void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Вижу, что OOPS!" );
        }
    }

    class ObsB
    {
        public static void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
    }

    class Class1
    {   static void Main()
        {
            Subj s  = new Subj();

            ObsA o1 = new ObsA();
            ObsA o2 = new ObsA();

            s.Oops += o1.OnOops;
            s.Oops += o2.OnOops;
            s.Oops += ObsB.OnOops;
            s.Oops += delegate ( object sender, EventArgs e )
                      { Console.WriteLine( "Я с вами!" ); };

            s.CryOops();
        }
    }
}
```

События включены во многие стандартные классы .NET, например, в классы пространства имен `Windows.Forms`, используемые для разработки Windows-приложений. Мы рассмотрим эти классы в главе 14.

Многопоточные приложения

Приложение .NET состоит из одного или нескольких *процессов*. Процессу принадлежат выделенная для него область оперативной памяти и ресурсы. Каждый процесс может состоять из нескольких *доменов* (частей) приложения, ресурсы

которых изолированы друг от друга. В рамках домена может быть запущено несколько потоков выполнения. *Поток* (thread[1]) представляет собой часть исполняемого кода программы. В каждом процессе есть *первичный поток*, исполняющий роль точки входа в приложение. Для консольных приложений это метод `Main`.

Многопоточные приложения создают как для многопроцессорных, так и для однопроцессорных систем. Основной целью при этом являются повышение общей производительности и сокращение времени реакции приложения. Управление потоками осуществляет операционная система. Каждый поток получает некоторое количество квантов времени, по истечении которого управление передается другому потоку. Это создает у пользователя однопроцессорной машины впечатление одновременной работы нескольких потоков и позволяет, к примеру, выполнять ввод текста одновременно с длительной операцией по передаче данных.

Недостатки многопоточности:

- большое количество потоков ведет к увеличению накладных расходов, связанных с их переключением, что снижает общую производительность системы;
- в многопоточных приложениях возникают проблемы синхронизации данных, связанные с потенциальной возможностью доступа к одним и тем же данным со стороны нескольких потоков (например, если один поток начинает изменение общих данных, а отведенное ему время истекает, доступ к этим же данным может получить другой поток, который, изменяя данные, необратимо их повреждает).

Класс Thread

Поддержка многопоточности осуществляется в .NET в основном с помощью пространства имен `System.Threading`. Некоторые типы этого пространства описаны в табл. 10.1.

Таблица 10.1. Некоторые типы пространства имен System.Threading

Тип	Описание
`Interlocked`	Класс, обеспечивающий синхронизированный доступ к переменным, которые используются в разных потоках
`Monitor`	Класс, обеспечивающий синхронизацию доступа к объектам
`Mutex`	Класс-примитив синхронизации, который используется также для синхронизации между процессами
`ReaderWriterLock`	Класс, определяющий блокировку, поддерживающую один доступ на запись и несколько — на чтение
`Thread`	Класс, который создает поток, устанавливает его приоритет, получает информацию о состоянии

[1] Иногда этот термин переводится буквально — «нить», чтобы отличить его от потоков ввода-вывода, которые рассматриваются в следующей главе. Поэтому в литературе можно встретить и термин «многонитевые приложения».

Тип	Описание
ThreadPool	Класс, используемый для управления набором взаимосвязанных потоков — пулом потоков
Timer	Класс, определяющий механизм вызова заданного метода в заданные интервалы времени для пула потоков
WaitHandle	Класс, инкапсулирующий объекты синхронизации, которые ожидают доступа к разделяемым ресурсам
IOCompletionCallback	Класс, получающий сведения о завершившейся операции ввода-вывода
ThreadStart	Делегат, представляющий метод, который должен быть выполнен при запуске потока
TimerCallback	Делегат, представляющий метод, обрабатывающий вызовы от класса Timer
WaitCallback	Делегат, представляющий метод для элементов класса ThreadPool
ThreadPriority	Перечисление, описывающее приоритет потока
ThreadState	Перечисление, описывающее состояние потока

Первичный поток создается автоматически. Для запуска вторичных потоков используется класс Thread. При создании объекта-потока ему передается делегат, определяющий метод, выполнение которого выделяется в отдельный поток:

```
Thread t = new Thread ( new ThreadStart( имя_метода ) );
```

После создания потока заданный метод начинает в нем свою работу, а первичный поток продолжает выполняться. В листинге 10.10 приведен пример одновременной работы двух потоков.

Листинг 10.10. Создание вторичного потока

```
using System;
using System.Threading;
namespace ConsoleApplication1
{    class Program
    {
        static public void Hedgehog()          // метод для вторичного потока
        {
            for ( int i = 0; i < 6; ++i )
            {
                Console.WriteLine( i ); Thread.Sleep( 1000 );
            }
        }

        static void Main()
        {
            Console.WriteLine( "Первичный поток " +
                Thread.CurrentThread.GetHashCode() );
```

продолжение

Листинг 10.10 (*продолжение*)

```
            Thread ta = new Thread( new ThreadStart(Hedgehog) );
            Console.WriteLine( "Вторичный поток " + ta.GetHashCode() );
            ta.Start();

            for ( int i = 0; i > -6; --i )
            {
                Console.Write( " " + i ); Thread.Sleep( 400 );
            }
        }
    }
}
```

Результат работы программы:

```
Первичный поток 1
Вторичный поток 2
 0 0 -1 -2 1 -3 -4 2 -5 3 4 5
```

В листинге используется метод Sleep, останавливающий функционирование потока на заданное количество миллисекунд. Как видите, оба потока работают одновременно. Если бы они работали с одним и тем же файлом, он был бы испорчен так же, как и приведенный вывод на консоль, поэтому такой способ распараллеливания вычислений имеет смысл только для работы с различными ресурсами.

В табл. 10.2 перечислены основные элементы класса Thread.

Таблица 10.2. Основные элементы класса Thread

Элемент	Вид	Описание
CurrentThread	Статическое свойство	Возвращает ссылку на выполняющийся поток (только для чтения)
IsAlive	Свойство	Возвращает true или false в зависимости от того, запущен поток или нет
IsBackground	Свойство	Возвращает или устанавливает значение, которое показывает, является ли этот поток фоновым
Name	Свойство	Установка текстового имени потока
Priority	Свойство	Получить/установить приоритет потока (используются значения перечисления ThreadPrority)
ThreadState	Свойство	Возвращает состояние потока (используются значения перечисления ThreadState)
Abort	Метод	Генерирует исключение ThreadAbortException. Вызов этого метода обычно завершает работу потока
GetData, SetData	Статические методы	Возвращает (устанавливает) значение для указанного слота в текущем потоке
GetDomain, GetDomainID	Статические методы	Возвращает ссылку на домен приложения (идентификатор домена приложения), в рамках которого работает поток

Элемент	Вид	Описание
GetHashCode	Метод	Возвращает хеш-код для потока
Sleep	Статический метод	Приостанавливает выполнение текущего потока на заданное количество миллисекунд
Interrupt	Метод	Прерывает работу текущего потока
Join	Метод	Блокирует вызывающий поток до завершения другого потока или указанного промежутка времени и завершает поток
Resume	Метод	Возобновляет работу после приостановки потока
Start	Метод	Начинает выполнение потока, определенного делегатом ThreadStart
Suspend	Метод	Приостанавливает выполнение потока. Если выполнение потока уже приостановлено, то игнорируется

Можно создать несколько потоков, которые будут совместно использовать один и тот же код. Пример приведен в листинге 10.11.

Листинг 10.11. Потоки, использующие один объект

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
    class Class1
    {   public void Do()
        {
            for ( int i = 0; i < 4; ++i )
                { Console.Write( " " + i ); Thread.Sleep( 3 ); }
        }
    }

    class Program
    {   static void Main()
        {
            Class1 a = new Class1();
            Thread t1 = new Thread( new ThreadStart( a.Do ) );
            t1.Name = "Second";
            Console.WriteLine( "Поток " + t1.Name );
            t1.Start();

            Thread t2 = new Thread( new ThreadStart( a.Do ) );
            t2.Name = "Third";
            Console.WriteLine( "Поток " + t2.Name );
            t2.Start();
        }
    }
}
```

Результат работы программы:

```
Поток Second
Поток Third
 0 0 1 1 2 2 3 3
```

Варианты вывода могут несколько различаться, поскольку один поток прерывает выполнение другого в неизвестные моменты времени.

Для того чтобы блок кода мог использоваться в каждый момент только одним потоком, применяется *оператор* `lock`. Формат оператора:

```
lock ( выражение ) блок_операторов
```

Выражение определяет объект, который требуется заблокировать. Для обычных методов в качестве выражения используется ключевое слово `this`, для статических — `typeof(класс)`. *Блок операторов* задает критическую секцию кода, которую требуется заблокировать.

Например, блокировка операторов в приведенном ранее методе `Do` выглядит следующим образом:

```
public void Do()
{
    lock( this )
    {
        for ( int i = 0; i < 4; ++i )
            { Console.Write( " " + i ); Thread.Sleep( 30 ); }
    }
}
```

Для такого варианта метода результат работы программы изменится:

```
Поток Second
Поток Third
 0 1 2 3 0 1 2 3
```

Асинхронные делегаты

Делегат можно вызвать на выполнение либо синхронно, как во всех приведенных ранее примерах, либо асинхронно с помощью методов `BeginInvoke` и `EndInvoke`.

При вызове делегата с помощью метода `BeginInvoke` среда выполнения создает для исполнения метода отдельный поток и возвращает управление оператору, следующему за вызовом. При этом в исходном потоке можно продолжать вычисления.

Если при вызове `BeginInvoke` был указан метод обратного вызова, этот метод вызывается после завершения потока. Метод обратного вызова также задается с помощью делегата, при этом используется стандартный делегат `AsyncCallback`. В методе обратного вызова для получения возвращаемого значения и выходных параметров применяется метод `EndInvoke`.

Если метод обратного вызова не был указан в параметрах метода BeginInvoke, метод EndInvoke можно использовать в потоке, инициировавшем запрос.

В листинге 10.11 приводятся два примера асинхронного вызова метода, выполняющего разложение числа на множители. Листинг приводится по документации Visual Studio с некоторыми изменениями.

Класс Factorizer содержит метод Factorize, выполняющий разложение на множители. Этот метод асинхронно вызывается двумя способами: в методе Num1 метод обратного вызова задается в BeginInvoke, в методе Num2 имеют место ожидание завершения потока и непосредственный вызов EndInvoke.

Листинг 10.11. Асинхронные делегаты

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;
                                                    // асинхронный делегат
public delegate bool AsyncDelegate ( int Num, out int m1, out int m2 );

                        // класс, выполняющий разложение числа на множители
public class Factorizer
{
    public bool Factorize( int Num, out int m1, out int m2 )
    {
        m1 = 1;    m2 = Num;
        for ( int i = 2; i < Num; i++ )
            if ( 0 == (Num % i) ) { m1 = i; m2 = Num / i; break; }

        if (1 == m1 ) return false;
        else          return true;
    }
}

                                    // класс, получающий делегата и результаты
public class PNum
{
    private int Number;
    public PNum( int number ) { Number = number; }

    [OneWayAttribute()]
                                              // метод, получающий результаты
    public void Res( IAsyncResult ar )
    {
        int m1, m2;
                                         // получение делегата из AsyncResult
        AsyncDelegate ad = (AsyncDelegate)((AsyncResult)ar).AsyncDelegate;

                        // получение результатов выполнения метода Factorize
        ad.EndInvoke( out m1, out m2, ar );
```

продолжение

Листинг 10.11 (*продолжение*)

```
                                                    // вывод результатов
        Console.WriteLine( "Первый способ : множители {0} : {1} {2}",
                           Number, m1, m2 );
    }
}
                                                    // демонстрационный класс
public class Simple
{
                          // способ 1: используется функция обратного вызова
    public void Num1()
    {
        Factorizer     f = new Factorizer();
        AsyncDelegate ad = new AsyncDelegate ( f.Factorize );

        int Num = 1000589023, tmp;
                        // создание экземпляра класса, который будет вызван
                                // после завершения работы метода Factorize
        PNum n = new PNum( Num );

                                // задание делегата метода обратного вызова
        AsyncCallback callback = new AsyncCallback( n.Res );

                                        // асинхронный вызов метода Factorize
        IAsyncResult ar = ad.BeginInvoke(
                              Num, out tmp, out tmp, callback, null );
        //
        // здесь - выполнение неких дальнейших действий
        // ...
    }
                       // способ 2: используется ожидание окончания выполнения
    public void Num2()
    {
        Factorizer     f = new Factorizer();
        AsyncDelegate ad = new AsyncDelegate ( f.Factorize );

        int Num = 1000589023, tmp;

                        // создание экземпляра класса, который будет вызван
                                // после завершения работы метода Factorize
        PNum n = new PNum( Num );

                                // задание делегата метода обратного вызова
        AsyncCallback callback = new AsyncCallback( n.Res );

                                        // асинхронный вызов метода Factorize
        IAsyncResult ar = ad.BeginInvoke(
                              Num, out tmp, out tmp, null, null );
                                                    // ожидание завершения
        ar.AsyncWaitHandle.WaitOne( 10000, false );
```

```
        if ( ar.IsCompleted )
        {
            int m1, m2;
                        // получение результатов выполнения метода Factorize
            ad.EndInvoke( out m1, out m2, ar );
                                                        // вывод результатов
            Console.WriteLine( "Второй способ : множители {0} : {1} {2}",
                               Num, m1, m2 );
        }
    }

    public static void Main()
    {
        Simple s = new Simple();
        s.Num1();
        s.Num2();
    }
}
```

Результат работы программы:

```
Первый способ : множители 1000589023 : 7 142941289
Второй способ : множители 1000589023 : 7 142941289
```

ПРИМЕЧАНИЕ

Атрибут `[OneWayAttribute()]` помечает метод как не имеющий возвращаемого значения и выходных параметров.

Рекомендации по программированию

Делегаты широко применяются в библиотеке .NET как самостоятельно, так и для поддержки механизма *событий*, который имеет важнейшее значение при программировании под Windows.

Делегат представляет собой особый вид класса, несколько напоминающий интерфейс, но, в отличие от него, задающий только одну сигнатуру метода. В языке C++ аналогом делегата является указатель на функцию, но он не обладает безопасностью и удобством использования делегата. Благодаря делегатам становится возможной гибкая организация взаимодействия, позволяющая поддерживать согласованное состояние взаимосвязанных объектов.

Начиная с версии 2.0, в C# поддерживаются возможности, упрощающие процесс программирования с применением делегатов — неявное создание делегатов при регистрации обработчиков событий и анонимные обработчики.

Основной целью создания *многопоточных приложений* является повышение общей производительности программы. Однако разработка многопоточных приложений сложнее, поскольку при этом возникают проблемы синхронизации данных, связанные с потенциальной возможностью доступа к одним и тем же данным со стороны нескольких потоков.

Глава 11
Работа с файлами

Под файлом обычно подразумевается именованная информация на внешнем носителе, например на жестком или гибком магнитном диске. Логически файл можно представить как конечное количество последовательных байтов, поэтому такие устройства, как дисплей, клавиатура и принтер, также можно рассматривать как частные случаи файлов. Передача данных с внешнего устройства в оперативную память называется *чтением*, или *вводом*, обратный процесс — *записью*, или *выводом*.

Ввод-вывод в C# выполняется с помощью подсистемы ввода-вывода и классов библиотеки .NET. В этой главе рассматривается обмен данными с файлами и их частным случаем — консолью. Обмен данными реализуется с помощью потоков.

Поток (stream)[1] — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Потоки обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис. Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер).

Обмен с потоком для повышения скорости передачи данных производится, как правило, через специальную область оперативной памяти — *буфер*. Буфер выделяется для каждого открытого файла. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока весь буфер не заполнится. Только после этого или после специальной команды сброса происходит передача данных на внешнее устройство. При чтении из файла данные вначале считываются в буфер, причем не столько, сколько запрашивается, а сколько помещается в буфер.

- Механизм буферизации позволяет более быстро и эффективно обмениваться информацией с внешними устройствами.

---

[1] Не путать с потоком выполнения, описанным в предыдущей главе. Впрочем, и в обычной речи мы часто обозначаем одним и тем же словом совершенно разные вещи!

Для поддержки потоков библиотека .NET содержит иерархию классов, основная часть которой представлена на рис. 11.1. Эти классы определены в пространстве имен `System.IO`. Помимо классов там описано большое количество перечислений для задания различных свойств и режимов.

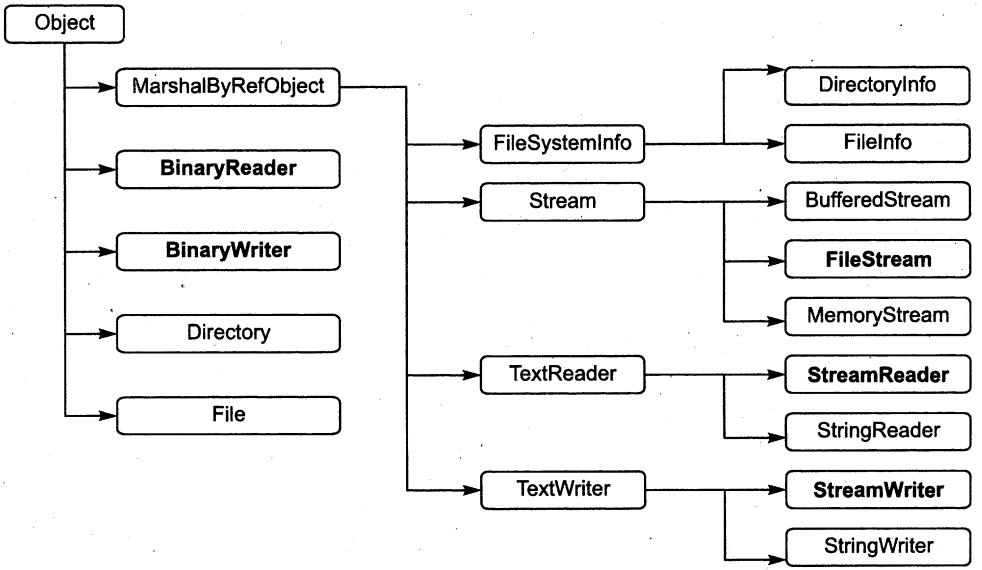


Рис. 11.1. Классы библиотеки .NET для работы с потоками

Классы библиотеки позволяют работать в различных режимах с файлами, каталогами и областями оперативной памяти. Краткое описание классов приведено в табл. 11.1.

Таблица 11.1. Основные классы пространства имен System.IO

Класс	Описание
`BinaryReader`, `BinaryWriter`	Чтение и запись значений простых встроенных типов (целочисленных, логических, строковых и т. п.) во внутренней форме представления
`BufferedStream`	Временное хранение потока байтов (например, для последующего переноса в постоянное хранилище)
`Directory`, `DirectoryInfo`, `File`, `FileInfo`	Работа с каталогами или физическими файлами: создание, удаление, получение свойств. Возможности классов `File` и `Directory` реализованы в основном в виде статических методов. Аналогичные классы `DirectoryInfo` и `FileInfo` используют обычные методы
`FileStream`	Произвольный (прямой) доступ к файлу, представленному как поток байтов
`MemoryStream`	Произвольный доступ к потоку байтов в оперативной памяти

продолжение

Таблица 11.1 (*продолжение*)

Класс	Описание
`StreamWriter`, `StreamReader`	Чтение из файла и запись в файл текстовой информации (произвольный доступ не поддерживается)
`StringWriter`, `StringReader`	Работа с текстовой информацией в оперативной памяти

Как можно видеть из таблицы, выполнять обмен с внешними устройствами можно на уровне:

- *двоичного представления данных* (`BinaryReader`, `BinaryWriter`);
- *байтов* (`FileStream`);
- *текста*, то есть символов (`StreamWriter`, `StreamReader`).

В .NET используется кодировка Unicode, в которой каждый символ кодируется двумя байтами. Классы, работающие с текстом, являются оболочками классов, использующих байты, и автоматически выполняют перекодирование из байтов в символы и обратно.

Двоичные и байтовые потоки хранят данные в том же виде, в котором они представлены в оперативной памяти, то есть при обмене с файлом происходит побитовое копирование информации. Двоичные файлы применяются не для просмотра их человеком, а для использования в программах.

Доступ к файлам может быть *последовательным*, когда очередной элемент можно прочитать (записать) только после аналогичной операции с предыдущим элементом, и *произвольным*, или *прямым*, при котором выполняется чтение (запись) произвольного элемента по заданному адресу. Текстовые файлы позволяют выполнять только последовательный доступ, в двоичных и байтовых потоках можно использовать оба метода.

Прямой доступ в сочетании с отсутствием преобразований обеспечивает высокую скорость получения нужной информации.

ПРИМЕЧАНИЕ

Методы *форматированного ввода*, с помощью которых можно выполнять ввод с клавиатуры или из текстового файла значений арифметических типов, в C# не поддерживаются. Для преобразования из символьного в числовое представление используются методы класса `Convert` или метод `Parse`, рассмотренные в разделе «Простейший ввод-вывод» (см. с. 59).

ПРИМЕЧАНИЕ

Форматированный вывод, то есть преобразование из внутренней формы представления числа в символьную, понятную человеку, выполняется с помощью перегруженных методов `ToString`, результаты выполнения которых передаются в методы текстовых файлов.

Помимо перечисленных классов в библиотеке .NET есть классы `XmlTextReader` и `XmlTextWriter`, предназначенные для формирования и чтения кода в формате XML. Понятие об XML дается в главе 15.

Рассмотрим простейшие способы работы с файловыми потоками. Использование классов файловых потоков в программе предполагает следующие операции:

1. Создание потока и связывание его с физическим файлом.
2. Обмен (ввод-вывод).
3. Закрытие файла.

Каждый класс файловых потоков содержит несколько вариантов конструкторов, с помощью которых можно создавать объекты этих классов различными способами и в различных режимах.

Например, файлы можно открывать только для чтения, только для записи или для чтения и записи. Эти *режимы доступа* к файлу содержатся в перечислении `FileAccess`, определенном в пространстве имен `System.IO`. Константы перечисления приведены в табл. 11.2.

Таблица 11.2. Значения перечисления FileAccess

Значение	Описание
`Read`	Открыть файл только для чтения
`ReadWrite`	Открыть файл для чтения и записи
`Write`	Открыть файл только для записи

Возможные *режимы открытия* файла определены в перечислении `FileMode` (табл. 11.3).

Таблица 11.3. Значения перечисления FileMode

Значение	Описание
`Append`	Открыть файл, если он существует, и установить текущий указатель в конец файла. Если файл не существует, создать новый файл
`Create`	Создать новый файл. Если в каталоге уже существует файл с таким же именем, он будет стерт
`CreateNew`	Создать новый файл. Если в каталоге уже существует файл с таким же именем, возникает исключение `IOException`
`Open`	Открыть существующий файл
`OpenOrCreate`	Открыть файл, если он существует. Если нет, создать файл с таким именем
`Truncate`	Открыть существующий файл. После открытия он должен быть обрезан до нулевой длины

Режим `FileMode.Append` можно использовать только совместно с доступом типа `FileAccess.Write`, то есть для файлов, открываемых для записи.

Режимы совместного использования файла различными пользователями определяет перечисление `FileShare` (табл. 11.4).

Таблица 11.4. Значения перечисления FileShare

Значение	Описание
None	Совместное использование открытого файла запрещено. Запрос на открытие данного файла завершается сообщением об ошибке
Read	Позволяет открывать файл для чтения одновременно нескольким пользователям. Если этот флаг не установлен, запросы на открытие файла для чтения завершаются сообщением об ошибке
ReadWrite	Позволяет открывать файл для чтения и записи одновременно нескольким пользователям
Write	Позволяет открывать файл для записи одновременно нескольким пользователям

Потоки байтов

Ввод-вывод в файл на уровне байтов выполняется с помощью класса FileStream, который является наследником абстрактного класса Stream, определяющего набор стандартных операций с потоками. Элементы класса Stream описаны в табл. 11.5.

Таблица 11.5. Элементы класса Stream

Элемент	Описание
BeginRead, BeginWrite	Начать асинхронный ввод или вывод
CanRead, CanSeek, CanWrite	Свойства, определяющие, какие операции поддерживает поток: чтение, прямой доступ и/или запись
Close	Закрыть текущий поток и освободить связанные с ним ресурсы (сокеты, указатели на файлы и т. п.)
EndRead, EndWrite	Ожидать завершения асинхронного ввода; закончить асинхронный вывод
Flush	Записать данные из буфера в связанный с потоком источник данных и очистить буфер. Если для данного потока буфер не используется, то этот метод ничего не делает
Length	Возвратить длину потока в байтах
Position	Возвратить текущую позицию в потоке
Read, ReadByte	Считать последовательность байтов (или один байт) из текущего потока и переместить указатель в потоке на количество считанных байтов
Seek	Установить текущий указатель потока на заданную позицию
SetLength	Установить длину текущего потока
Write, WriteByte	Записать последовательность байтов (или один байт) в текущий поток и переместить указатель в потоке на количество записанных байтов

Класс FileStream реализует эти элементы для работы с дисковыми файлами. Для определения режимов работы с файлом используются стандартные перечисления FileMode, FileAccess и FileShare. Значения этих перечислений приведены в табл. 11.2—11.4. В листинге 11.1 представлен пример работы с файлом. В примере демонстрируются чтение и запись одного байта и массива байтов, а также позиционирование в потоке.

Листинг 11.1. Пример использования потока байтов

```
using System;
using System.IO;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            FileStream f = new FileStream( "test.txt",
                FileMode.Create, FileAccess.ReadWrite );

            f.WriteByte( 100 );       // в начало файла записывается число 100

            byte[] x = new byte[10];
            for ( byte i = 0; i < 10; ++i )
            {
                x[i] = (byte)( 10 - i );
                f.WriteByte(i);             // записывается 10 чисел от 0 до 9
            }

            f.Write( x, 0, 5 );            // записывается 5 элементов массива

            byte[] y = new byte[20];

            f.Seek( 0, SeekOrigin.Begin );    // текущий указатель - на начало
            f.Read( y, 0, 20 );                    // чтение из файла в массив

            foreach ( byte elem in y ) Console.Write( " " + elem );
            Console.WriteLine();

            f.Seek(5, SeekOrigin.Begin); // текущий указатель - на 5-й элемент
            int a = f.ReadByte();                      // чтение 5-го элемента
            Console.WriteLine( a );
            a = f.ReadByte();                          // чтение 6-го элемента
            Console.WriteLine( a );

            Console.WriteLine( "Текущая позиция в потоке" + f.Position );
            f.Close();
        }
    }
}
```

Результат работы программы:

```
 100 0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 0 0 0 0
4
5
Текущая позиция в потоке 7
```

Текущая позиция в потоке первоначально устанавливается на начало файла (для любого режима открытия, кроме Append) и сдвигается на одну позицию при записи каждого байта.

Для установки желаемой позиции чтения используется метод Seek, имеющий два параметра: первый задает смещение в байтах относительно точки отсчета, задаваемой вторым. Точки отсчета задаются константами перечисления SeekOrigin: начало файла — Begin, текущая позиция — Current и конец файла — End.

В данном примере файл создавался в текущем каталоге. Можно указать и полный путь к файлу, при этом удобнее использовать дословные литералы, речь о которых шла в разделе «Литералы» (см. с. 30), например:

```
FileStream f = new FileStream( @"D:\C#\test.txt",
              FileMode.Create, FileAccess.ReadWrite );
```

В дословных литералах не требуется дублировать обратную косую черту.

Операции по открытию файлов могут завершиться неудачно, например, при ошибке в имени существующего файла или при отсутствии свободного места на диске, поэтому *рекомендуется всегда контролировать результаты* этих операций.

В случае непредвиденных ситуаций среда выполнения генерирует различные исключения, обработку которых следует предусмотреть в программе, например:

- FileNotFoundException, если файла с указанным именем в указанном каталоге не существует;
- DirectoryNotFoundException, если не существует указанный каталог;
- ArgumentException, если неверно задан режим открытия файла;
- IOException, если файл не открывается из-за ошибок ввода-вывода.

Возможны и другие исключительные ситуации.

Удобно обрабатывать наиболее вероятные ошибки раздельно, чтобы предоставить пользователю программы в выводимом сообщении наиболее точную информацию. В приведенном далее примере отдельно перехватывается ошибка в имени файла, а затем обрабатываются все остальные возможные ошибки:

```
try
{
    FileStream f = new FileStream( @"d:\C#\test.tx",
        FileMode.Open, FileAccess.Read );
    ...                                              // действия с файлом
    f.Close();
}

catch( FileNotFoundException e )
```

```
{
    Console.WriteLine( e.Message );
    Console.WriteLine( "Проверьте правильность имени файла!" );
    return;
}

catch( Exception e )
{
    Console.WriteLine( "Error: " + e.Message );
    return;
}
```

При закрытии файла освобождаются все связанные с ним ресурсы, например, для файла, открытого для записи, в файл выгружается содержимое буфера. Поэтому рекомендуется всегда закрывать файлы после окончания работы, в особенности файлы, открытые для записи. Если буфер требуется выгрузить, не закрывая файл, используется метод `Flush`.

Асинхронный ввод-вывод

Класс `Stream` (и, соответственно, `FileStream`) поддерживает два способа выполнения операций ввода-вывода: синхронный и асинхронный. По умолчанию файлы открываются в синхронном режиме, то есть последующие операторы выполняются только после завершения операций ввода-вывода. Для длительных файловых операций более эффективно выполнять ввод-вывод асинхронно, в отдельном потоке выполнения[1]. При этом в первичном потоке можно выполнять другие операции.

Для асинхронного ввода-вывода необходимо открыть файл в асинхронном режиме, для этого используется соответствующий вариант перегруженного конструктора. Асинхронная операция ввода инициируется с помощью метода `BeginRead`. Помимо характеристик буфера, в который выполняется ввод, в этот метод передается делегат, задающий метод, выполняемый после завершения ввода.

Этот метод может инициировать обработку полученной информации, возобновить операцию чтения или выполнить любые другие действия, например, проверить успешность ввода и сообщить о его завершении. Обычно в этом методе вызывается метод `EndRead`, который завершает асинхронную операцию.

Аналогично выполняется и асинхронный вывод. В листинге 11.2 приведен пример асинхронного чтения из файла большого объема и параллельного выполнения диалога с пользователем.

ПРИМЕЧАНИЕ

Вообще говоря, существуют различные способы завершения асинхронных операций, и здесь демонстрируется только один из них.

[1] Работа с потоками выполнения рассматривалась в предыдущей главе.

Листинг 11.2. Асинхронный ввод

```
using System;
using System.IO;
using System.Threading;

namespace ConsoleApplication1
{
    class Demo
    {
        public void UserInput()                    // диалог с пользователем
        {
            string s;
            do
            {   Console.WriteLine( "Введите строку, Enter для завершения" );
                s = Console.ReadLine();
            } while (s.Length != 0 );
        }

        public void  OnCompletedRead( IAsyncResult ar )                  // 1
        {
            int bytes = f.EndRead( ar );
            Console.WriteLine( "Считано " + bytes );
        }

        public void AsyncRead()
        {
            f = new FileStream( "D:\\verybigfile", FileMode.Open,
                FileAccess.Read, FileShare.Read, buf.Length, true );     // 2

            callback = new AsyncCallback( OnCompletedRead );             // 3

            f.BeginRead( buf, 0, buf.Length, callback, null );           // 4
        }

        FileStream f;
        byte[] buf = new byte[66666666];
        AsyncCallback callback;
    }

    class Program
    {   static void Main()
        {
            Demo d = new Demo();
            d.AsyncRead();
            d.UserInput();
        }
    }
}
```

Для удобства восприятия операции чтения из файла и диалога с пользователем оформлены в отдельный класс Demo.

Метод OnCompletedRead (оператор 1) должен получать один параметр стандартного типа IAsyncResult, содержащий сведения о завершении операции, которые передаются в метод EndRead.

Файл открывается в асинхронном режиме, об этом говорит значение true последнего параметра конструктора (оператор 2). В операторе 3 создается экземпляр стандартного делегата AsyncCallback, который инициализируется методом OnCompletedRead.

С помощью этого делегата метод OnCompletedRead передается в метод BeginRead (оператор 4), который создает отдельный поток, начинает асинхронный ввод и возвращает управление в вызвавший поток. Обратный вызов метода OnCompletedRead происходит при завершении операции ввода. При достаточно длинном файле verybigfile можно убедиться, что приглашение к вводу в методе UserInput выдается раньше, чем сообщение о завершении операции ввода из метода OnCompletedRead.

ПРИМЕЧАНИЕ

Пример, приведенный в листинге 11.2, максимально упрощен для демонстрации методов BeginRead и EndRead, поэтому в нем нет необходимых в любой программе проверок наличия файла, успешности считывания и т. д.

Потоки символов

Символьные потоки StreamWriter и StreamReader работают с Unicode-символами[1], следовательно, ими удобнее всего пользоваться для работы с файлами, предназначенными для восприятия человеком. Эти потоки являются наследниками классов TextWriter и TextReader соответственно, которые обеспечивают их большей частью функциональности. В табл. 11.6 и 11.7 приведены наиболее важные элементы этих классов. Как видите, произвольный доступ для текстовых файлов не поддерживается.

Таблица 11.6. Наиболее важные элементы базового класса TextWriter

Элемент	Описание
Close	Закрыть файл и освободить связанные с ним ресурсы. Если в процессе записи используется буфер, он будет автоматически очищен
Flush	Очистить все буферы для текущего файла и записать накопленные в них данные в место их постоянного хранения. Сам файл при этом не закрывается

продолжение

[1] Существует возможность изменить используемую кодировку с помощью объекта System.Text.Encoding.

Таблица 11.6 (*продолжение*)

Элемент	Описание
NewLine	Используется для задания последовательности символов, означающих начало новой строки. По умолчанию используется последовательность «возврат каретки» — «перевод строки» (\r\n)
Write	Записать фрагмент текста в поток
WriteLine	Записать строку в поток и перейти на другую строку

Таблица 11.7. Наиболее важные элементы класса TextReader

Элемент	Описание
Peek	Возвратить следующий символ, не изменяя позицию указателя в файле
Read	Считать данные из входного потока
ReadBlock	Считать из входного потока указанное пользователем количество символов и записать их в буфер, начиная с заданной позиции
ReadLine	Считать строку из текущего потока и возвратить ее как значение типа string. Пустая строка (null) означает конец файла (EOF)
ReadToEnd	Считать все символы до конца потока, начиная с текущей позиции, и возвратить считанные данные как одну строку типа string

Вы уже знакомы с некоторыми методами, приведенными в этих таблицах: на протяжении всей книги постоянно использовались методы чтения из текстовых потоков и записи в текстовые потоки, но не для дисковых файлов, а для консоли, которая является их частным случаем.

В листинге 11.3 создается текстовый файл, в который записываются две строки. Вторая строка формируется из преобразованных численных значений переменных и поясняющего текста. Содержимое файла можно посмотреть в любом текстовом редакторе. Файл создается в том же каталоге, куда среда записывает исполняемый файл. По умолчанию это каталог ...\ConsoleApplication1\bin\Debug.

Листинг 11.3. Вывод в текстовый файл

```
using System;
using System.IO;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            try
            {
                StreamWriter f = new StreamWriter( "text.txt" );

                f.WriteLine( "Вывод в текстовый файл:" );

                double a = 12.234;
```

```
            int    b = 29;
            f.WriteLine( " a = {0,6:C}  b = {1,2:X}", a, b );

            f.Close();

        }

        catch( Exception e )
        {
            Console.WriteLine( "Error: " + e.Message );
            return;
        }
    }
}
```

В листинге 11.4 файл, созданный в предыдущем листинге, выводится на экран.

Листинг 11.4. Чтение текстового файла

```
using System;
using System.IO;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            try
            {
                StreamReader f = new StreamReader( "text.txt" );

                string s = f.ReadToEnd();
                Console.WriteLine(s);

                f.Close();
            }

            catch( FileNotFoundException e )
            {
                Console.WriteLine( e.Message );
                Console.WriteLine( " Проверьте правильность имени файла!" );
                return;
            }

            catch( Exception e )
            {
                Console.WriteLine( "Error: " + e.Message );
                return;
            }
        }
    }
}
```

В этой программе весь файл считывается за один прием с помощью метода ReadToEnd. Чаще возникает необходимость считывать файл построчно, такой пример приведен в листинге 11.5. Каждая строка при выводе предваряется номером.

Листинг 11.5. Построчное чтение текстового файла

```
using System;
using System.IO;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            try
            {
                StreamReader f = new StreamReader( "text.txt" );

                string s;
                long i = 0;

                while ( ( s = f.ReadLine() ) != null )
                     Console.WriteLine( "{0}: {1}", ++i, s );
                f.Close();
            }

            catch( FileNotFoundException e )
            {
                Console.WriteLine( e.Message );
                Console.WriteLine( "Проверьте правильность имени файла!" );
                return;
            }

            catch( Exception e )
            {
                Console.WriteLine( "Error: " + e.Message );
                return;
            }
        }
    }
}
```

Пример преобразования чисел, содержащихся в текстовом файле, в их внутреннюю форму представления приведен в листинге 11.6. В программе вычисляется сумма чисел в каждой строке.

На содержимое файла накладываются весьма строгие ограничения: числа должны быть разделены ровно одним пробелом, после последнего числа в строке пробела быть не должно, файл не должен заканчиваться символом перевода строки. Методы разбиения строки и преобразования в целочисленное представление рассматривались ранее.

Листинг 11.6. Преобразования строк в числа

```
using System;
using System.IO;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            try
            {
                StreamReader f = new StreamReader( "numbers.txt" );

                string s;

                const int n = 20;
                int[] a = new int[n];
                string[] buf;

                while ( ( s = f.ReadLine() ) != null )
                {
                    buf = s.Split(' ');
                    long sum = 0;
                    for ( int i = 0; i < buf.Length; ++i )
                    {
                        a[i] = Convert.ToInt32( buf[i] );
                        sum += a[i];
                    }
                    Console.WriteLine( "{0}  сумма: {1}", s, sum );
                }
                f.Close();
            }

            catch( FileNotFoundException e )
            {
                Console.WriteLine( e.Message );
                Console.WriteLine( " Проверьте правильность имени файла!" );
                return;
            }

            catch( Exception e )
            {
                Console.WriteLine( "Error: " + e.Message );
                return;
            }
        }
    }
}
```

Результат работы программы:

```
1 2 4  сумма: 7
3 44 -3 6  сумма: 50
8 1 1  сумма: 10
```

Двоичные потоки

Двоичные файлы хранят данные в том же виде, в котором они представлены в оперативной памяти, то есть во внутренней форме представления. Двоичные файлы применяются не для просмотра их человеком, а для использования в программах. Выходной поток `BinaryWriter` поддерживает произвольный доступ, то есть имеется возможность выполнять запись в произвольную позицию двоичного файла.

Двоичный файл открывается на основе базового потока, в качестве которого чаще всего используется поток `FileStream`. Входной двоичный поток содержит перегруженные методы чтения для всех простых встроенных типов данных.

Основные методы двоичных потоков приведены в табл. 11.8 и 11.9.

Таблица 11.8. Наиболее важные элементы класса BinaryWriter

Элемент	Описание
`BaseStream`	Базовый поток, с которым работает объект `BinaryWriter`
`Close`	Закрыть поток
`Flush`	Очистить буфер
`Seek`	Установить позицию в текущем потоке
`Write`	Записать значение в текущий поток

Таблица 11.9. Наиболее важные элементы класса BinaryReader

Элемент	Описание
`BaseStream`	Базовый поток, с которым работает объект `BinaryReader`
`Close`	Закрыть поток
`PeekChar`	Возвратить следующий символ без перемещения внутреннего указателя в потоке
`Read`	Считать поток байтов или символов и сохранить в массиве, передаваемом как входной параметр
`ReadXXXX`	Считать из потока данные определенного типа (например, `ReadBoolean`, `ReadByte`, `ReadInt32` и т. д.)

В листинге 11.7 приведен пример формирования двоичного файла. В файл записывается последовательность вещественных чисел, а затем для демонстрации произвольного доступа третье число заменяется числом 8888.

Листинг 11.7. Формирование двоичного файла

```
using System;
using System.IO;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
```

```
        {
            try
            {
                BinaryWriter fout = new BinaryWriter(
                    new FileStream( @"D:\C#\binary", FileMode.Create) );

                double d = 0;
                while ( d < 4 )
                {
                    fout.Write( d );
                    d += 0.33;
                };

                fout.Seek( 16, SeekOrigin.Begin ); // второй элемент файла
                fout.Write( 8888d );

                fout.Close();
            }

            catch( Exception e )
            {
                Console.WriteLine( "Error: " + e.Message );
                return;
            }
        }
    }
}
```

При создании двоичного потока в него передается объект базового потока. При установке указателя текущей позиции в файле учитывается длина каждого значения типа `double` — 8 байт.

Попытка просмотра сформированного программой файла в текстовом редакторе весьма медитативная, но не информативная, поэтому в листинге 11.8 приводится программа, которая с помощью экземпляра `BinaryReader` считывает содержимое файла в массив вещественных чисел, а затем выводит этот массив на экран.

При чтении принимается во внимание тот факт, что метод `ReadDouble` при обнаружении конца файла генерирует исключение `EndOfStreamException`. Поскольку в данном случае это не ошибка, тело обработчика исключений пустое.

Листинг 11.8. Считывание двоичного файла

```
using System;
using System.IO;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            try
```

продолжение ⇩

Листинг 11.8 (*продолжение*)

```
        {
            FileStream f =
                new FileStream( @"D:\C#\binary", FileMode.Open );
            BinaryReader fin = new BinaryReader( f );

            long n = f.Length / 8;          // количество чисел в файле
            double[] x = new double[n];

            long i = 0;
            try
            {
                while( true ) x[i++] = fin.ReadDouble();       // чтение
            }
            catch ( EndOfStreamException e ) {}

            foreach( double d in x ) Console.Write( "  " + d ); // вывод

            fin.Close();
            f.Close();
        }

        catch ( FileNotFoundException e )
        {
            Console.WriteLine( e.Message );
            Console.WriteLine( "Проверьте правильность имени файла!" );
            return;
        }

        catch ( Exception e )
        {
            Console.WriteLine( "Error: " + e.Message );
            return;
        }
    }
  }
}
```

Результат работы программы:

```
0  0,33  8888  0,99  1,32  1,65  1,98  2,31  2,64  2,97  3,3  3,63  3,96
```

Консольный ввод-вывод

Консольные приложения имеют весьма ограниченную область применения, самой распространенной из которых является обучение языку программирования. Для организации ввода и вывода используется известный вам класс `Console`, определенный в пространстве имен `System`. В этом классе определены три стандартных

потока: входной поток Console.In класса TextReader и выходные потоки Console.Out и Console.Error класса TextWriter.

По умолчанию входной поток связан с клавиатурой, а выходные — с экраном, однако можно перенаправить эти потоки на другие устройства с помощью методов SetIn и SetOut или средствами операционной системы (перенаправление с помощью операций <, > и >>).

При обмене с консолью можно применять методы указанных потоков, но чаще используются методы класса Console — Read, ReadLine, Write и WriteLine, которые просто передают управление методам нижележащих классов In, Out и Error.

Использование не одного, а двух выходных потоков полезно при желании разделить нормальный вывод программы и ее сообщения об ошибках. Например, нормальный вывод программы можно перенаправить в файл, а сообщения об ошибках — на консоль или в файл журнала.

Работа с каталогами и файлами

В пространстве имен System.IO есть четыре класса, предназначенные для работы с физическими файлами и структурой каталогов на диске: Directory, File, DirectoryInfo и FileInfo. С их помощью можно выполнять создание, удаление, перемещение файлов и каталогов, а также получение их свойств.

Классы Directory и File реализуют свои функции через статические методы. DirectoryInfo и FileInfo обладают схожими возможностями, но они реализуются путем создания объектов соответствующих классов. Классы DirectoryInfo и FileInfo происходят от абстрактного класса FileSystemInfo, который снабжает их базовыми свойствами, описанными в табл. 11.10.

Таблица 11.10. Свойства класса FileSystemInfo

Свойство	Описание
Attributes	Получить или установить атрибуты для данного объекта файловой системы. Для этого свойства используются значения перечисления FileAttributes
CreationTime	Получить или установить время создания объекта файловой системы
Exists	Определить, существует ли данный объект файловой системы
Extension	Получить расширение файла
FullName	Возвратить имя файла или каталога с указанием полного пути
LastAccessTime	Получить или установить время последнего обращения к объекту файловой системы
LastWriteTime	Получить или установить время последнего внесения изменений в объект файловой системы
Name	Возвратить имя файла. Это свойство доступно только для чтения. Для каталогов возвращает имя последнего каталога в иерархии, если это возможно. Если нет, возвращает полностью определенное имя

Класс DirectoryInfo содержит элементы, позволяющие выполнять необходимые действия с каталогами файловой системы. Эти элементы перечислены в табл. 11.11.

Таблица 11.11. Элементы класса DirectoryInfo

Элемент	Описание
Create, CreateSubDirectory	Создать каталог или подкаталог по указанному пути в файловой системе
Delete	Удалить каталог со всем его содержимым
GetDirectories	Возвратить массив строк, представляющих все подкаталоги
GetFiles	Получить файлы в текущем каталоге в виде массива объектов класса FileInfo
MoveTo	Переместить каталог и все его содержимое на новый адрес в файловой системе
Parent	Возвратить родительский каталог

В листинге 11.9 приведен пример, в котором создаются два каталога, выводится информация о них и предпринимается попытка удаления каталога.

Листинг 11.9. Использование класса DirectoryInfo

```
using System;
using System.IO;
namespace ConsoleApplication1
{   class Class1
    {   static void DirInfo( DirectoryInfo di )
        {
            // Вывод информации о каталоге
            Console.WriteLine( "===== Directory Info =====" );
            Console.WriteLine( "FullName: "   + di.FullName );
            Console.WriteLine( "Name: "       + di.Name );
            Console.WriteLine( "Parent: "     + di.Parent );
            Console.WriteLine( "Creation: "   + di.CreationTime );
            Console.WriteLine( "Attributes: " + di.Attributes );
            Console.WriteLine( "Root: "       + di.Root );
            Console.WriteLine( "==========================" );
        }

        static void Main()
        {
            DirectoryInfo di1 = new DirectoryInfo( @"c:\MyDir" );
            DirectoryInfo di2 = new DirectoryInfo( @"c:\MyDir\temp" );
            try
            {
                // Создать каталоги
                di1.Create();
```

```
            di2.Create();

            // Вывести информацию о каталогах
            DirInfo(di1);
            DirInfo(di2);

            // Попытаться удалить каталог
            Console.WriteLine( "Попытка удалить {0}.", di1.Name );
            di1.Delete();
        }

        catch ( Exception )
        {
            Console.WriteLine( "Попытка не удалась " );
        }
    }
  }
}
```

Результат работы программы:

```
===== Directory Info =====
FullName: c:\MyDir
Name: MyDir
Parent:
Creation: 30.04.2006 17:14:44
Attributes: Directory
Root: c:\
==========================
===== Directory Info =====
FullName: c:\MyDir\temp
Name: temp
Parent: MyDir
Creation: 30.04.2006 17:14:44
Attributes: Directory
Root: c:\
==========================
Попытка удалить MyDir.
Попытка не удалась
```

Каталог не пуст, поэтому попытка его удаления не удалась. Впрочем, если использовать перегруженный вариант метода `Delete` с одним параметром, задающим режим удаления, можно удалить и непустой каталог:

```
di1.Delete( true );                    // удаляет непустой каталог
```

Обратите внимание на свойство `Attributes`. Некоторые его возможные значения, заданные в перечислении `FileAttributes`, приведены в табл. 11.12.

Таблица 11.12. Некоторые значения перечисления FileAttributes

Значение	Описание
Archive	Используется приложениями при выполнении резервного копирования, а в некоторых случаях — при удалении старых файлов
Compressed	Файл является сжатым
Directory	Объект файловой системы является каталогом
Encrypted	Файл является зашифрованным
Hidden	Файл является скрытым
Normal	Файл находится в обычном состоянии, и для него установлены любые другие атрибуты. Этот атрибут не может использоваться с другими атрибутами
Offline	Файл, расположенный на сервере, кэширован в хранилище на клиентском компьютере. Возможно, что данные этого файла уже устарели
ReadOnly	Файл доступен только для чтения
System	Файл является системным

Листинг 11.10 демонстрирует использование класса FileInfo для копирования всех файлов с расширением jpg из каталога d:\foto в каталог d:\temp. Метод Exists позволяет проверить, существует ли исходный каталог.

Листинг 11.10. Копирование файлов

```
using System;
using System.IO;
namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            try
            {
                string DestName = @"d:\temp\";
                DirectoryInfo dest = new DirectoryInfo( DestName );
                dest.Create();                  // создание целевого каталога

                DirectoryInfo dir = new DirectoryInfo( @"d:\foto" );

                if ( ! dir.Exists )      // проверка существования каталога
                {
                    Console.WriteLine( "Каталог " +
                                       dir.Name + " не существует" );
                    return;
                }

                FileInfo[] files = dir.GetFiles( "*.jpg" ); //список файлов
```

```
            foreach( FileInfo f in files )
                f.CopyTo( dest + f.Name );          // копирование файлов

            Console.WriteLine( "Скопировано " +
                               files.Length + " jpg-файлов" );
        }

        catch ( Exception e )
        {
            Console.WriteLine( "Error: " + e.Message );
        }
    }
  }
}
```

Использование классов `File` и `Directory` аналогично, за исключением того, что их методы являются статическими и, следовательно, не требуют создания объектов.

Сохранение объектов (сериализация)

В C# есть возможность сохранять на внешних носителях не только данные примитивных типов, но и объекты. Сохранение объектов называется *сериализацией*, а восстановление сохраненных объектов — *десериализацией*. При сериализации объект преобразуется в линейную последовательность байтов. Это сложный процесс, поскольку объект может включать множество унаследованных полей и ссылки на вложенные объекты, которые, в свою очередь, тоже могут состоять из объектов сложной структуры.

К счастью, сериализация выполняется автоматически, достаточно просто пометить класс как сериализуемый с помощью атрибута `[Serializable]`. Атрибуты рассматриваются в главе 12, пока же достаточно знать, что атрибуты — это дополнительные сведения о классе, которые сохраняются в его метаданных. Те поля, которые сохранять не требуется, помечаются атрибутом `[NonSerialized]`, например:

```
[Serializable]
class Demo
{
    public int a = 1;
    [NonSerialized]
    public double y;
    public Monster X, Y;
}
```

Объекты можно сохранять в одном из двух форматов: двоичном или SOAP (в виде XML-файла). В первом случае следует подключить к программе пространство имен `System.Runtime.Serialization.Formatters.Binary`, во втором — пространство `System.Runtime.Serialization.Formatters.Soap`.

Рассмотрим *сохранение объектов в двоичном формате*. Для этого используется класс BinaryFormatter, в котором определены два метода:

```
Serialize( поток, объект );
Deserialize( поток );
```

Метод Serialize сохраняет заданный объект в заданном потоке, метод Deserialize восстанавливает объект из заданного потока.

В листинге 11.11 объект приведенного ранее класса Demo сохраняется в файле на диске с именем Demo.bin. Этот файл можно просмотреть, открыв его, к примеру, в Visual Studio.NET.

Листинг 11.11. Сериализация объекта

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
namespace ConsoleApplication1
{
    [Serializable]
    abstract class Spirit
    {
        public abstract void Passport();
    }

    [Serializable]
    class Monster : Spirit
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }

        override public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        string name;
        int health, ammo;
    }

    [Serializable]
    class Demo
    {
        public int a = 1;
```

```
    [NonSerialized]
    public double b;
    public Monster X, Y;
}

class Class1
{   static void Main()
    {
        Demo d = new Demo();
        d.X = new Monster( 100, 80, "Вася" );
        d.Y = new Monster( 120, 50, "Петя" );
        d.a = 2;
        d.b = 2;

        d.X.Passport();            d.Y.Passport();
        Console.WriteLine( d.a );  Console.WriteLine( d.b );

        FileStream f = new FileStream( "Demo.bin", FileMode.Create );
        BinaryFormatter bf = new BinaryFormatter();

        bf.Serialize( f, d );          // сохранение объекта d в потоке f

        f.Close();
    }
}
}
```

В программе не приведены неиспользуемые методы и свойства классов. Обратите внимание на то, что базовые классы сохраняемых объектов также должны быть помечены как сохраняемые. Результат работы программы:

```
Monster Вася     health = 100 ammo = 80
Monster Петя     health = 120 ammo = 50
2
2
```

Итак, *для сохранения объекта в двоичном формате необходимо*:

1. Подключить к программе пространство имен `System.Runtime.Serialization.Formatters.Binary`.
2. Пометить сохраняемый класс и связанные с ним классы атрибутом `[Serializable]`.
3. Создать поток и связать его с файлом на диске или с областью оперативной памяти.
4. Создать объект класса `BinaryFormatter`.
5. Сохранить объекты в потоке.
6. Закрыть файл.

В листинге 11.12 сохраненный объект считывается из файла.

Листинг 11.12. Десериализация объекта

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
namespace ConsoleApplication1
{
    ...
    class Class1
    {
        static void Main()
        {
            FileStream f = new FileStream( "Demo.bin", FileMode.Open );
            BinaryFormatter bf = new BinaryFormatter();
            Demo d = (Demo) bf.Deserialize( f );   // восстановление объекта

            d.X.Passport();              d.Y.Passport();
            Console.WriteLine( d.a );  Console.WriteLine( d.b );

            f.Close();
        }
    }
}
```

Результат работы программы:

```
Monster Вася     health = 100 ammo = 80
Monster Петя     health = 120 ammo = 50
2
0
```

Как видите, при сериализации сохраняется все дерево объектов. Обратите внимание на то, что значение поля y не было сохранено, поскольку оно было помечено как несохраняемое.

ПРИМЕЧАНИЕ

Сериализация в формате SOAP выполняется аналогично с помощью класса `SoapFormatter`. Программист может задать собственный формат сериализации, для этого ему придется реализовать в своих классах интерфейс `ISerializable` и специальный вид конструктора класса.

Рекомендации по программированию

Большинство программ тем или иным образом работают с внешними устройствами, в качестве которых могут выступать, например, консоль, файл на диске или сетевое соединение. Взаимодействие с внешними устройствами организуется с помощью потоков, которые поддерживаются множеством классов библиотеки .NET.

Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен. Классы библиотеки позволяют работать с потоками в различных режимах и на различных уровнях: на уровне двоичного представления данных, байтов и текста. Двоичные и байтовые потоки хранят данные во внутреннем представлении, текстовые — в кодировке Unicode.

Поток можно открыть в синхронном или асинхронном режиме для чтения, записи или добавления. Доступ к файлам может быть последовательным и произвольным. Текстовые файлы позволяют выполнять только последовательный доступ, в двоичных и байтовых потоках можно использовать оба метода. Прямой доступ в сочетании с отсутствием преобразований обеспечивает высокую скорость обмена.

Методы форматированного ввода для значений арифметических типов в C# не поддерживаются. Для преобразования из символьного в числовое представление используются методы класса `Convert` или метод `Parse`. Форматированный вывод выполняется с помощью перегруженного метода `ToString`, результат выполнения которого передается в методы текстовых файлов.

Рекомендуется всегда проверять успешность открытия существующего файла, перехватывать исключения, возникающие при преобразовании значений арифметических типов, и явным образом закрывать файл, в который выполнялась запись. Длительные операции с файлами более эффективно выполнять в асинхронном режиме.

Для сохранения объектов (сериализации) используется атрибут `[Serializable]`. Объекты можно сохранять в одном из двух форматов: двоичном или SOAP (в виде XML-файла).

Глава 12
Сборки, библиотеки, атрибуты, директивы

Все наши предыдущие приложения состояли из одного физического файла. Для больших проектов это неудобно и чаще всего невозможно, да и в других случаях бывает удобнее поместить связанные между собой типы в библиотеку и использовать их по мере необходимости. В этой главе мы рассмотрим вопросы создания и использования библиотек, способы получения и подготовки информации о типах, пространства имен и препроцессор. Эти сведения необходимы для успешной разработки реальных программ.

Сборки

В результате компиляции в среде .NET создается *сборка* — файл с расширением exe или dll, который содержит код на промежуточном языке, метаданные типов, манифест и ресурсы (рис. 12.1). Понятие сборки было введено в главе 1 (см. с. 9), а сейчас мы рассмотрим ее составные части более подробно.

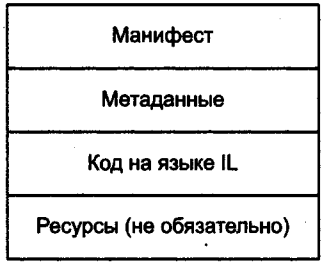


Рис. 12.1. Сборка, состоящая из одного файла

Промежуточный язык (Intermediate Language, IL) не содержит инструкций, зависящих от операционной системы и типа компьютера, что обеспечивает две основные возможности:

- выполнение приложения на любом типе компьютера, для которого существует среда выполнения CLR;
- повторное использование кода, написанного на любом .NET-совместимом языке.

IL-код можно просмотреть с помощью дизассемблера ILDasm.exe, который находится в папке ...\SDK\bin\ каталога размещения Visual Studio.NET. После запуска ILDasm можно открыть любой файл среды .NET с расширением exe или dll с помощью команды File ▸ Open. В окне программы откроется список всех элементов сборки, сведения о каждом можно получить двойным щелчком. При этом открывается окно, в котором для методов выводится доступный для восприятия дизассемблированный код.

Метаданные типов — это сведения о типах, используемых в сборке. Компилятор создает метаданные автоматически. В них содержится информация о каждом типе, имеющемся в программе, и о каждом его элементе. Например, для каждого класса описываются все его поля, методы, свойства, события, базовые классы и интерфейсы.

Среда выполнения использует метаданные для поиска определений типов и их элементов в сборке, для создания экземпляров объектов, проверки вызова методов и т. д. Компилятор, редактор кода и средства отладки также широко используют метаданные, например, для вывода подсказок и диагностических сообщений.

Манифест — это набор метаданных о самой сборке, включая информацию обо всех файлах, входящих в состав сборки, версии сборки, а также сведения обо всех внешних сборках, на которые она ссылается. Манифест создается компилятором автоматически, программист может дополнять его собственными атрибутами.

Чаще всего сборка состоит из единственного файла, однако она может включать и несколько физических файлов (модулей). В этом случае манифест либо включается в состав одного из файлов, либо содержится в отдельном файле. Многофайловые сборки используются для ускорения загрузки приложения — это имеет смысл для сборок большого объема, работа с которыми производится удаленно.

На логическом уровне сборка представляет собой совокупность взаимосвязанных типов — классов, интерфейсов, структур, перечислений, делегатов и ресурсов. Библиотека .NET представляет собой совокупность сборок, которую используют приложения. Точно так же можно создавать и собственные сборки, которые можно будет задействовать либо в рамках одного приложения (*частные сборки*), либо совместно различными приложениями (*открытые сборки*). По умолчанию все сборки являются частными.

Манифест сборки содержит:

- идентификатор версии;
- список всех внутренних модулей сборки;
- список внешних сборок, необходимых для нормального выполнения сборки;
- информацию о естественном языке, используемом в сборке (например, русском);
- «сильное» имя (strong name) — специальный вариант имени сборки, используемый для открытых сборок;
- необязательную информацию, связанную с безопасностью;
- необязательную информацию, связанную с хранением ресурсов внутри сборки (подробнее о форматах ресурсов .NET см. [27]).

Идентификатор версии относится ко всем элементам сборки. Он позволяет избегать конфликтов имен и поддерживать одновременное существование и использование различных версий одних и тех же сборок. Идентификатор версии состоит из двух частей: *информационной версии* в виде текстовой строки и *версии совместимости* в виде четырех чисел, разделенных точками:

- основной номер версии (major version);
- дополнительный номер версии (minor version);
- номер сборки (build number);
- номер ревизии (revision number).

Среда выполнения применяет идентификатор версий для определения того, какие из открытых сборок совместимы с требованиями клиента. Например, если клиент запрашивает сборку 3.1.0.0, а присутствует только версия 3.4.0.0, сборка не будет опознана как подходящая, поскольку считается, что в дополнительных версиях могут произойти изменения в типах и их элементах. Разные номера ревизии допускают, но не гарантируют совместимость. Номер сборки на совместимость не влияет, так как чаще всего он изменяется при установке *заплатки*, или *патча* (patch).

Идентификатор версии формируется автоматически, но при желании можно задать его вручную с помощью атрибута `[AssemblyVersion]`, который рассматривается далее на с. 285.

Информация о безопасности позволяет определить, предоставить ли клиенту доступ к запрашиваемым элементам сборки. В манифесте сборки определены ограничения системы безопасности.

Ресурсы представляют собой, например, файлы изображений, помещаемых на форму, текстовые строки, значки приложения и т. д. Хранение ресурсов внутри сборки обеспечивает их защиту и упрощает развертывание приложения. Среда Visual Studio.NET предоставляет возможности автоматического внедрения ресурсов в сборку.

Открытые и частные сборки различаются по способам размещения на компьютере пользователя, именованию и политике версий. Частные сборки должны находиться в каталоге приложения, использующего сборку, или в его подкаталогах.

Открытые сборки размещаются в специальном каталоге, который называется глобальным кэшем сборок (Global Assembly Cache, GAC). Для идентификации открытой сборки используется уже упоминавшееся *сильное имя* (strong name), которое должно быть уникальным.

Создание библиотеки

Для создания библиотеки следует при разработке проекта в среде Visual Studio.NET выбрать шаблон Class Library (библиотека классов). В главе 8 была создана простая иерархия классов персонажей компьютерной игры. В этом разделе мы оформим ее в виде библиотеки, то есть сборки с расширением dll. Для сборки задано имя MonsterLib (рис. 12.2).

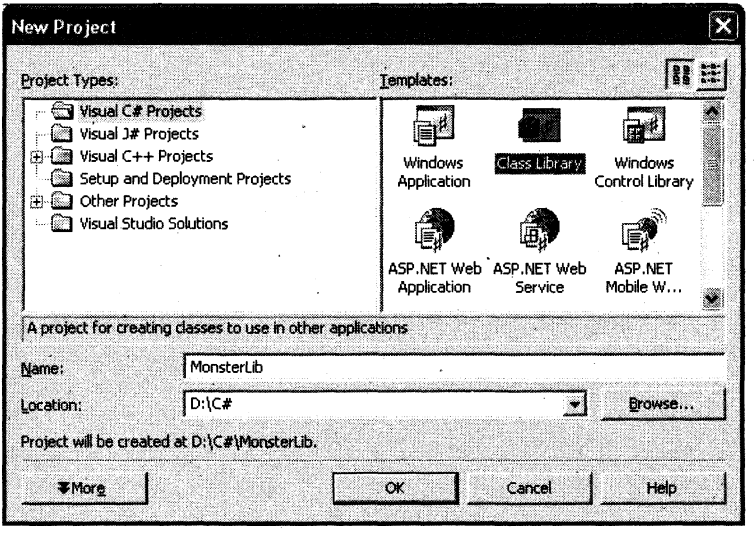


Рис. 12.2. Создание библиотеки

Текст модуля приведен в листинге 12.1. По сравнению с модулем из главы 8 в него добавлены спецификаторы доступа `public` для всех трех классов, входящих в библиотеку.

Листинг 12.1. Библиотека монстров

```
namespace MonsterLib
{
    using System;

    public abstract class Spirit
    {
        public abstract void Passport();
    }
```

продолжение ⇩

Листинг 12.1 (*продолжение*)

```
public class Monster : Spirit
{
    public Monster()
    {
        this.health = 100;
        this.ammo   = 100;
        this.name   = "Noname";
    }

    public Monster( string name ) : this()
    {
        this.name = name;
    }

    public Monster( int health, int ammo, string name )
    {
        this.health = health;
        this.ammo   = ammo;
        this.name   = name;
    }

    public int Health
    {
        get
        {
            return health;
        }
        set
        {
            if ( value > 0 ) health = value;
            else             health = 0;
        }
    }

    public int Ammo
    {
        get
        {
            return ammo;
        }
        set
        {
            if ( value > 0 ) ammo = value;
            else             ammo = 0;
        }
    }

    public string Name
    {
```

```
            get
            {
                return name;
            }
        }

        override public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        string name;
        int health, ammo;
    }

    public class Daemon : Monster
    {
        public Daemon()
        {
            brain = 1;
        }

        public Daemon( string name, int brain ) : base( name )
        {
            this.brain = brain;
        }

        public Daemon( int health, int ammo, string name, int brain ) :
            base( health, ammo, name )
        {
            this.brain = brain;
        }

        override public void Passport()
        {
            Console.WriteLine(
                "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
                 Name, Health, Ammo, brain );
        }

        public void Think()
        {
            Console.Write( Name + " is" );
            for ( int i = 0; i < brain; ++i ) Console.Write( " thinking" );
            Console.WriteLine( "..." );
        }

        int brain;
    }
}
```

Скомпилировав библиотеку, вы обнаружите файл MonsterLib.dll в каталогах ...\bin\Debug и ...\obj\Debug. Открыв файл MonsterLib.dll с помощью программы ILDasm.exe, можно получить полную информацию о созданной библиотеке (рис. 12.3).

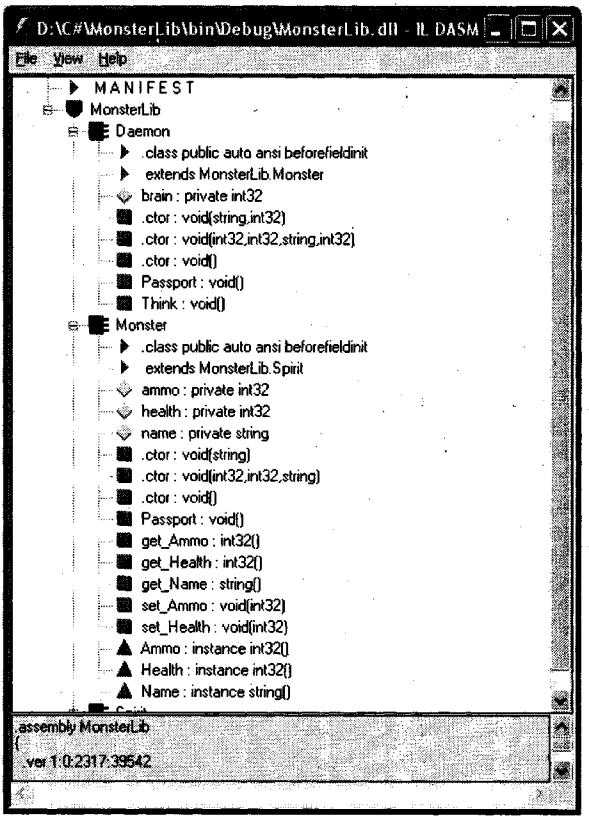


Рис. 12.3. Просмотр библиотеки с помощью дизассемблера ILDasm.exe

Использование библиотеки

Любая библиотека — это сервер, предоставляющий свои ресурсы клиентам. Создадим клиентское приложение, выполняющее те же функции, что и приложение из раздела «Виртуальные методы» (см. с. 178), но с использованием библиотеки MonsterLib.dll. Для того чтобы компилятор мог ее обнаружить, необходимо после создания проекта (как обычно, это — консольное приложение) подключить ссылку на библиотеку с помощью команды Project ▸ Add Reference (Добавить ссылку). Для поиска каталога, содержащего библиотеку, следует использовать кнопку Browse.

После подключения библиотеки можно пользоваться ее открытыми элементами таким же образом, как если бы они были описаны в том же модуле. Текст приложения приведен в листинге 12.2.

Листинг 12.2. Клиентское приложение

```
using System;
namespace ConsoleApplication1
{
    using MonsterLib;

    class Class1
    {   static void Main()
        {
            const int n = 3;
            Monster[] stado = new Monster[n];

            stado[0] = new Monster( "Monia" );
            stado[1] = new Monster( "Monk" );
            stado[2] = new Daemon ( "Dimon", 3 );

            foreach ( Monster elem in stado ) elem.Passport();

            for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;
            Console.WriteLine();

            foreach ( Monster elem in stado ) elem.Passport();
        }
    }
}
```

Результаты работы программы совпадают с полученными в листинге 8.3. Анализ каталога ...\bin\Debug показывает, что среда создала в нем копию библиотеки MonsterLib.dll, то есть поместила библиотеку в тот же каталог, что и исполняемый файл. Если скопировать эти два файла в другое место, программа не потеряет своей работоспособности — главное, чтобы оба файла находились в одном каталоге.

Допускается также, чтобы частные сборки находились в подкаталогах основного каталога приложения.

ПРИМЕЧАНИЕ

Преимущество .NET состоит в том, что благодаря стандартным соглашениям можно использовать библиотеки независимо от языка, на котором они были написаны. Таким образом, можно было бы написать клиентское приложение, например, на языке VB.NET.

Рефлексия

Рефлексия — это получение информации о типах во время выполнения программы. Например, можно получить список всех классов и интерфейсов сборки, список элементов каждого из классов, список параметров каждого метода и т. д. Вся информация берется из метаданных сборки. Для использования рефлексии необходимы класс `System.Type` и типы пространства имен `System.Reflection`.

В классе Type описаны методы, которые позволяют получить информацию о типах. В пространстве имен System.Reflection описаны типы, поддерживающие Type, а также классы, которые служат для организации позднего связывания и динамической загрузки сборок.

Наиболее важные свойства и методы класса Type приведены в табл. 12.1.

Таблица 12.1. Элементы класса Type

Элемент	Описание
IsAbstract, IsArray, IsNestedPublic, IsClass, IsNestedPrivate, IsCOMObject, IsEnum, IsInterface, IsPrimitive, IsSealed, IsValueType	Свойства, позволяющие получить соответствующие характеристики конкретного типа в программе (например, является ли он абстрактным, является ли он массивом, классом и т. п.). Приведены не все свойства
GetConstructors, GetEvents, GetFields, GetInterfaces, GetMethods, GetMembers, GetNestedTypes, GetProperties	Методы, возвращающие массив с набором соответствующих элементов (конструкторов, событий, полей и т. п.). Возвращаемое значение соответствует имени метода, например, GetFields возвращает массив типа FieldInfo, GetMethods — массив типа MethodInfo. Для каждого из методов есть парный ему (без символа s в конце имени), который предназначен для работы с одним заданным в параметре элементом (например, GetMethod и GetMethods)
FindMembers	Метод возвращает массив типа MemberInfo на основе заданных критериев поиска
GetType	Метод возвращает объект типа Type по имени, заданному в виде строки
InvokeMember	Метод используется для позднего связывания заданного элемента

Воспользоваться этими методами можно после создания экземпляра класса Type. Поскольку это абстрактный класс, обычный способ создания объектов с помощью операции new неприменим, зато существуют три других способа:

1. В базовом классе object описан метод GetType, которым можно воспользоваться для любого объекта, поскольку он наследуется. Метод возвращает объект типа Type, например:

```
Monster X = new Monster();
Type t = X.GetType();
```

2. В классе Type описан статический метод GetType с одним параметром строкового типа, на место которого требуется передать имя класса (типа), например:

```
Type t = Type.GetType( "Monster" );
```

3. Операция typeof возвращает объект класса Type для типа, заданного в качестве параметра, например:

```
Type t = typeof( Monster );
```

При использовании второго и третьего способов создавать экземпляр исследуемого класса нет необходимости.

Как видно из табл. 12.1, многие методы класса Type возвращают экземпляры стандартных классов (например, MemberInfo). Эти классы описаны в пространстве имен System.Reflection. Наиболее важные из этих классов перечислены в табл. 12.2.

Таблица 12.2. Некоторые классы пространства имен System.Reflection

Тип	Описание
Assembly	Содержит методы для получения информации о сборке, а также для загрузки сборки и выполнения с ней различных операций
AssemblyName	Позволяет получать информацию о сборке (имя, версия, совместимость, естественный язык и т. п.)
EventInfo	Хранит информацию о событии
FieldInfo	Хранит информацию о поле
MemberInfo	Абстрактный базовый класс, определяющий общие элементы для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo
MethodInfo	Хранит информацию о методе
Module	Позволяет обратиться к модулю в многофайловой сборке
ParameterInfo	Хранит информацию о параметре
PropertyInfo	Хранит информацию о свойстве

В листинге 12.3 приведены примеры использования рассмотренных методов и классов для получения информации о классах библиотеки из листинга 12.1.

Листинг 12.3. Получение информации о классах

```
using System;
using System.Reflection;
namespace ConsoleApplication1
{
    using MonsterLib;

    class Class1
    {   static void Info( Type t )
        {
            Console.WriteLine( "======= Класс " + t.FullName );
            if ( t.IsAbstract ) Console.WriteLine( "абстрактный" );
            if ( t.IsClass    ) Console.WriteLine( "обычный" );
            if ( t.IsEnum     ) Console.WriteLine( "перечисление" );

            Console.WriteLine( "базовый класс " + t.BaseType );
            MethodInfo[] met = t.GetMethods();
            foreach ( MethodInfo m in met ) Console.WriteLine( m );
            Console.WriteLine();
```

продолжение

Листинг 12.3 (*продолжение*)

```
            PropertyInfo[] prs = t.GetProperties();
            foreach ( PropertyInfo p in prs ) Console.WriteLine( p );
            Console.WriteLine();
        }

        static void Main()
        {
            Type t = typeof( Spirit );
            Info( t );

            t = typeof( Monster );
            Info( t );

            t = typeof( Daemon );
            Info( t );
        }
    }
}
```

Результат работы программы:

```
======= Класс MonsterLib.Spirit
абстрактный
обычный
базовый класс System.Object
Void Passport()
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
System.Type GetType()

======= Класс MonsterLib.Monster
обычный
базовый класс MonsterLib.Spirit
Void Passport()
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Int32 get_Health()
Void set_Health(Int32)
Int32 get_Ammo()
Void set_Ammo(Int32)
System.String get_Name()
System.Type GetType()

Int32 Health
Int32 Ammo
System.String Name
```

```
======= Класс MonsterLib.Daemon
обычный
базовый класс MonsterLib.Monster
Void Passport()
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Think()
Int32 get_Health()
Void set_Health(Int32)
Int32 get_Ammo()
Void set_Ammo(Int32)
System.String get_Name()
System.Type GetType()

Int32 Health
Int32 Ammo
System.String Name
```

Можно продолжить исследования дальше, например, получить параметры и возвращаемое значение каждого метода. Думаю, что принцип вам уже ясен. Напомню, что вся эта информация берется из метаданных сборки.

Атрибуты

Атрибуты — это дополнительные сведения об элементах программы (классах, методах, параметрах и т. д.). С помощью атрибутов можно добавлять информацию в метаданные сборки и затем извлекать ее во время выполнения программы. Атрибут является специальным видом класса и происходит от базового класса `System.Attribute`.

Атрибуты делятся на стандартные и пользовательские. В библиотеке .NET предусмотрено множество стандартных атрибутов, которые можно использовать в программах. Если всего разнообразия стандартных атрибутов не хватит, чтобы удовлетворить прихотливые требования программиста, он может описать собственные классы атрибутов, после чего применять их точно так же, как стандартные.

При *использовании* (*спецификации*) *атрибутов* они задаются в *секции атрибутов*, располагаемой непосредственно перед элементом, для описания которого они предназначены. Секция заключается в квадратные скобки и может содержать несколько атрибутов, перечисляемых через запятую. Порядок следования атрибутов произвольный.

Для каждого атрибута задаются *имя*, а также необязательные *параметры* и *тип элемента сборки*, к которому относится атрибут. Простейший пример атрибута:

```
[Serializable]
class Monster
{
    ...
```

```
    [NonSerialized]
    string name;
    int health, ammo;
}
```

Атрибут [Serializable], означающий, что объекты этого класса можно сохранять во внешней памяти, относится ко всему классу Monster. При этом поле name помечено атрибутом [NonSerialized], что говорит о том, что это поле сохраняться не должно. Сохранение объектов рассматривалось в главе 10.

Обычно из контекста понятно, к какому элементу сборки относится атрибут, однако в некоторых случаях могут возникнуть неоднозначности. Для их устранения перед именем атрибута записывается *тип элемента сборки* — уточняющее ключевое слово, отделяемое от атрибута двоеточием. Ключевые слова и соответствующие элементы сборки, к которым могут относиться атрибуты, перечислены в табл. 12.3.

Таблица 12.3. Типы элемента сборки, задаваемые для атрибутов

Ключевое слово	Описание
assembly	Атрибут относится ко всей сборке
field	Атрибут относится к полю
event	Атрибут относится к событию
method	Атрибут относится к методу
param	Атрибут относится к параметрам метода
property	Атрибут относится к свойству
return	Атрибут относится к возвращаемому значению
type	Атрибут относится к классу или структуре

Пусть, например, перед методом описан гипотетический атрибут ABC:

```
[ABC]
public void Do()  { ... }
```

По умолчанию он относится к методу. Чтобы указать, что атрибут относится не к методу, а к его возвращаемому значению, следует написать:

```
[return:ABC]
public void Do()  { ... }
```

Атрибут может иметь *параметры*. Они записываются в круглых скобках через запятую после имени атрибута и бывают *позиционными* и *именованными*. Именованный параметр указывается в форме имя = значение, для позиционного просто задается значение. Например, для использованного в следующем фрагменте кода атрибута CLSCompliant задан позиционный параметр true. Атрибуты,

относящиеся к сборке, должны располагаться непосредственно после директив using, например:

```
using System;
[assembly:CLSCompliant(true)]
namespace ConsoleApplication1
{ ...
```

Атрибут [CLSCompliant] определяет, удовлетворяет программный код соглашениям CLS (Common Language Specification) или нет.

Стандартные атрибуты, как и другие типы классов, имеют набор конструкторов, которые определяют, каким образом использовать (специфицировать) атрибут. Фактически, при использовании атрибута указывается наиболее подходящий конструктор, а величины, не указанные в конструкторе, задаются через именованные параметры в конце списка параметров.

Стандартный атрибут [STAThread], старательно удаленный из всех листингов в этой книге, относится к методу, перед которым он записан. Он имеет значение только для приложений, использующих модель COM, и задает модель потоков в рамках модели COM. Пример применения еще одного стандартного атрибута, [Conditional], приведен далее в разделе «Директивы препроцессора».

Атрибуты уровня сборки хранятся в файле AssemblyInfo.cs, автоматически создаваемом средой для любого проекта. Для явного задания номера версии сборки можно записать атрибут [AssemblyVersion], например:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Создание пользовательских атрибутов выходит за рамки темы этого учебника. Интересующиеся могут обратиться к книге [27].

Пространства имен

Пространство имен — это контейнер для типов, определяющий область их видимости. Пространства имен предотвращают конфликты имен и используются для двух взаимосвязанных целей:

- логического группирования элементов программы, расположенных в различных физических файлах;
- группирования имен, предоставляемых сборкой в пользование другим модулям.

Во всех программах, созданных ранее, использовалось пространство имен, создаваемое по умолчанию. Реальные программы чаще всего разрабатываются группой программистов, каждый из которых работает со своим набором физических файлов (единиц компиляции), хранящих элементы создаваемого приложения. Если в разных файлах описать пространства имен с одним и тем же именем, то при построении приложения, состоящего из этих файлов, будет скомпоновано единое пространство имен.

Пространства имен могут быть вложенными, например:

```
namespace State
{
    namespace City
    { ...
    }
}
```

Такое объявление аналогично следующему:

```
namespace State.City
{ ...
}
```

Вложенные пространства имен, как вы наверняка успели заметить, широко применяются в библиотеке .NET.

Существует *три способа использования типа*, определенного в каком-либо пространстве имен:

1. Использовать полностью квалифицированное имя. Например, в пространстве имен `System.Runtime.Serialization.Formatters.Binary` описан класс `BinaryFormatter`. Создание объекта этого класса с помощью квалифицированного имени выглядит так:

```
System.Runtime.Serialization.Formatters.Binary.BinaryFormatter bf =
  new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
```

2. Использовать *директиву* `using`, с помощью которой импортируются все имена из заданного пространства. В этом случае предыдущий пример примет вид

```
using System.Runtime.Serialization.Formatters.Binary;
...
BinaryFormatter bf = new BinaryFormatter();
```

ВНИМАНИЕ

Директива `using` должна располагаться вне или внутри пространства имен, но до любых описаний типов.

3. Использовать *псевдоним типа*. Это делается с помощью второй формы директивы `using`:

```
using BinF =
  System.Runtime.Serialization.Formatters.Binary.BinaryFormatter;
...
BinF bf = new BinF();
```

Первый способ применяется при однократном использовании имени типа из «неглубоко» вложенных пространств имен, второй — в большинстве остальных случаев, что мы и делали во всех примерах, а третий можно рекомендовать при многократном использовании длинного имени типа.

В версию языка C# 2.0 введена возможность применять *псевдоним пространства имен* с помощью операции ::, например:

```
using SIO = System.IO;                          // псевдоним пространства имен
using MIO = MyLibrary.IO;                       // псевдоним пространства имен
class Program
{
    static void Main() {
        SIO::Stream s = new MIO::EmptyStream(); // использование псевдонимов
        ...
    }
}
```

Использование псевдонима для пространства имен гарантирует, что последующие подключения других пространств имен к этой сборке не повлияют на существующие определения. Слева от операции :: можно указать идентификатор `global`. Он гарантирует, что поиск идентификатора, расположенного справа от операции, будет выполняться только в глобальном пространстве имен. Цель использования этого идентификатора та же: не допустить изменений существующих определений при разработке следующих версий программы, в которых в нее могут быть добавлены новые пространства имен, содержащие элементы с такими же именами.

Таким образом, сборки обеспечивают физическое группирование типов, а пространства имен — логическое. В мире сетевого программирования, когда программисту доступны десятки тысяч классов, пространства имен совершенно необходимы как для классификации и поиска, так и для предотвращения конфликтов имен типов.

Директивы препроцессора

Препроцессором в языке C++ называется предварительный этап компиляции, формирующий окончательный вариант текста программы. В языке C#, потомке C++, препроцессор практически отсутствует, но некоторые директивы сохранились. Назначение директив — исключать из процесса компиляции фрагменты кода при выполнении определенных условий, выводить сообщения об ошибках и предупреждения, а также структурировать код программы.

Каждая директива располагается на отдельной строке и *не* заканчивается точкой с запятой, в отличие от операторов языка. В одной строке с директивой может располагаться только комментарий вида //. Перечень и краткое описание директив приведены в табл. 12.4.

Рассмотрим более подробно применение директив условной компиляции. Они используются для того, чтобы исключить компиляцию отдельных частей программы. Это бывает полезно при отладке или, например, при поддержке нескольких версий программы для различных платформ.

Таблица 12.4. Директивы препроцессора

Наименование	Описание
#define, #undef	Определение (например, #define DEBUG) и отмена определения (#undef DEBUG) символьной константы, которая используется директивами условной компиляции. Директивы размещаются до первой лексемы единицы компиляции. Допускается повторное определение одной и той же константы
#if, #elif, #else, #endif	Директивы условной компиляции. Код, находящийся в области их действия, компилируется или пропускается в зависимости от того, была ли ранее определена символьная константа (см. далее)
#line	Задание номера строки и имени файла, о котором выдаются сообщения, возникающие при компиляции (например, #line 200 "ku_ku.txt"). При этом в диагностических сообщениях компилятора имя компилируемого файла заменяется указанным, а строки нумеруются, начиная с заданного первым параметром номера[1]
#error, #warning	Вывод *при компиляции* сообщения, указанного в строке директивы. После выполнения директивы #error компиляция прекращается (например, #error Дальше компилировать нельзя). После выполнения директивы #warning компиляция продолжается
#region, #endregion	Определение фрагмента кода, который можно будет свернуть или развернуть средствами редактора кода. Фрагмент располагается между этими директивами
#pragma	Введена в версии C# 2.0. Позволяет отключить (#pragma warning disable) или восстановить (#pragma warning restore) выдачу всех или перечисленных в директиве предупреждений компилятора

Формат директив:

```
#if константное_выражение
...
[ #elif константное_выражение
...]
[ #elif константное_выражение
...]
[ #else
...]
#endif
```

Количество директив #elif произвольно. Исключаемые блоки кода могут содержать как описания, так и исполняемые операторы. *Константное выражение* может содержать одну или несколько символьных констант, объединенных знаками операций ==, !=, !, && и ||. Также допускаются круглые скобки. Константа считается равной true, если она была ранее определена с помощью директивы #define.

[1] «...Натуре находится много вещей, неизъяснимых даже для обширного ума». — Н. В. Гоголь.

Пример применения директив приведен в листинге 12.4.

Листинг 12.4. Применение директив условной компиляции

```
// #define VAR1
// #define VAR2
using System;
namespace ConsoleApplication1
{
    class Class1
    {
#if VAR1
        static void F(){ Console.WriteLine( "Вариант 1" ); }
#elif VAR2
        static void F(){ Console.WriteLine( "Вариант 2" ); }
#else
        static void F(){ Console.WriteLine( "Основной вариант" ); }
#endif
        static void Main()
        {
            F();
        }
    }
}
```

В зависимости от того, определение какой символьной константы раскомментировать, в компиляции будет участвовать один из трех методов F.

Директива #define применяется не только в сочетании с директивами условной компиляции. Можно применять ее вместе со стандартным атрибутом Conditional для условного управления выполнением методов. Метод будет выполняться, если константа определена. Пример приведен в листинге 12.5. Обратите внимание на то, что для применения атрибута необходимо подключить пространство имен System.Diagnostics.

Листинг 12.5. Использование атрибута Conditional

```
// #define VAR1
 #define VAR2
using System;
using System.Diagnostics;
namespace ConsoleApplication1
{
    class Class1
    {
[Conditional ("VAR1")]
        static void A(){ Console.WriteLine( "Выполняется метод A" ); }

[Conditional ("VAR2")]
```

продолжение ⇩

Листинг 12.5 (*продолжение*)

```
        static void B(){ Console.WriteLine( "Выполняется метод B" ); }

        static void Main()
        {
            A(); B();
        }
    }
}
```

В методе `Main` записаны вызовы обоих методов, однако в данном случае будет выполнен только метод `B`, поскольку символьная константа `VAR1` не определена.

Рекомендации по программированию

В этой главе приведено краткое введение в средства, использующиеся при профессиональной разработке программ: библиотеки, рефлексия типов, атрибуты, пространства имен и директивы препроцессора. Для реального использования этих возможностей в программах необходимо изучать документацию и дополнительную литературу, например [20], [21], [26], [27], [30].

Глава 13
Структуры данных, коллекции и классы-прототипы

В этой главе приводится обзор основных структур данных, используемых в программировании, и их реализация в библиотеке .NET в виде коллекций. Кроме того, описываются средства, добавленные в версию C# 2.0, — классы-прототипы (generics), частичные типы (partial types) и обнуляемые типы (nullable types)[1].

Абстрактные структуры данных

Любая программа предназначена для обработки данных, от способа организации которых зависит ее алгоритм. Для разных задач необходимы различные способы хранения и обработки данных, поэтому выбор структур данных должен предшествовать созданию алгоритмов и основываться на требованиях к функциональности и быстродействию программы.

Наиболее часто в программах используются следующие структуры данных: массив, список, стек, очередь, бинарное дерево, хеш-таблица, граф и множество. Далее даны краткие характеристики каждой из этих структур.

Массив — это конечная совокупность однотипных величин. Массив занимает непрерывную область памяти и предоставляет прямой, или произвольный, доступ к своим элементам по индексу. Память под массив выделяется до начала работы с ним и впоследствии не изменяется.

В *списке* каждый элемент связан со следующим и, возможно, с предыдущим. В первом случае список называется *односвязным*, во втором — *двусвязным*. Также применяются термины *однонаправленный* и *двунаправленный*. Если последний элемент

---

[1] Остальные средства, появившиеся в версии 2;0, были рассмотрены ранее. Например, итераторы — в главе 9, анонимные методы — в главе 10.

связать указателем с первым, получается *кольцевой список*. Количество элементов в списке может изменяться в процессе работы программы.

Каждый элемент списка содержит *ключ*, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью данных, хранящихся в каждом элементе списка. В качестве ключа в процессе работы со списком могут выступать разные части данных. Например, если создается список из записей, содержащих фамилию, год рождения, стаж работы и пол, любая часть записи может выступать в качестве ключа: при упорядочивании списка по алфавиту ключом будет фамилия, а при поиске, например, ветеранов труда ключом можно сделать стаж. Ключи разных элементов списка могут совпадать.

Над списками можно выполнять следующие операции:

- добавление элемента в конец списка;
- чтение элемента с заданным ключом;
- вставка элемента в заданное место списка;
- удаление элемента с заданным ключом;
- упорядочивание списка по ключу.

Список не обеспечивает произвольный доступ к элементу, поэтому при выполнении операций чтения, вставки и удаления выполняется последовательный перебор элементов, пока не будет найден элемент с заданным ключом. Для списков большого объема перебор элементов может занимать значительное время, поскольку среднее время поиска элемента пропорционально количеству элементов в списке.

Стек — это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO (Last In — First Out, последним пришел — первым ушел). Стек проще всего представить себе как закрытую с одного конца узкую трубу, в которую бросают мячики. Достать первый брошенный мячик можно только после того, как вынуты все остальные. Стеки широко применяются в системном программном обеспечении, компиляторах, в различных рекурсивных алгоритмах.

Очередь — это частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (First In — First Out, первым пришел — первым ушел). Очередь проще всего представить себе, постояв в ней час-другой. В программировании очереди применяются, например, в моделировании, диспетчеризации задач операционной системой, буферизованном вводе-выводе.

Бинарное дерево — это динамическая структура данных, состоящая из узлов, каждый из которых содержит помимо данных не более двух ссылок на различные бинарные поддеревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется *корнем* дерева.

Пример бинарного дерева приведен на рис. 13.1 (корень обычно изображается сверху). Узел, не имеющий поддеревьев, называется *листом*. Исходящие узлы называются *предками*, входящие — *потомками*. *Высота дерева* определяется количеством уровней, на которых располагаются его узлы.

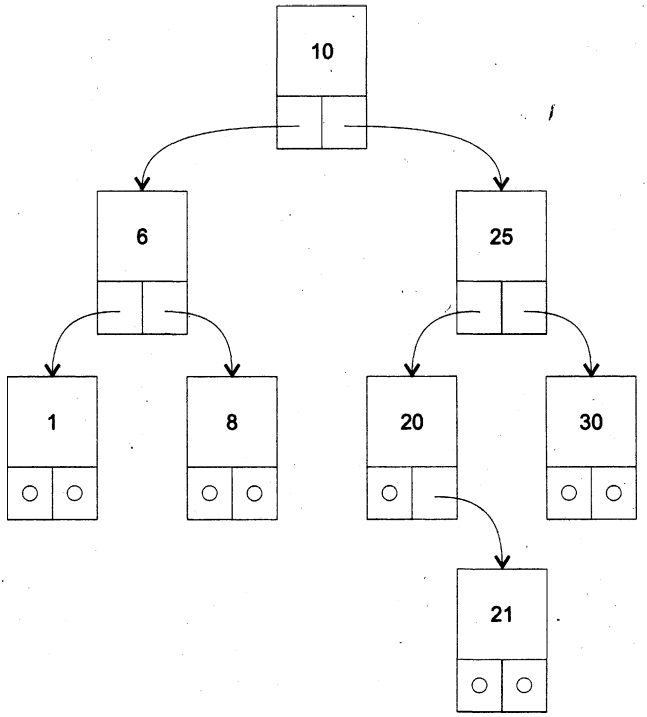


Рис. 13.1. Пример бинарного дерева поиска

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется *деревом поиска*. Одинаковые ключи не допускаются.

В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму количества узлов. Впрочем, скорость поиска в значительной степени зависит от порядка формирования дерева: если на вход подается упорядоченная или почти упорядоченная последовательность ключей, дерево вырождается в список. Для ускорения поиска применяется процедура балансировки дерева, формирующая дерево, поддеревья которого различаются не более чем на один элемент. Бинарные деревья применяются для эффективного поиска и сортировки данных.

Хеш-таблица, *ассоциативный массив*, или *словарь* — это массив, доступ к элементам которого осуществляется не по номеру, а по некоторому ключу. Можно

сказать, что это таблица, состоящая из пар «ключ—значение» (табл. 13.1). Хеш-таблица эффективно реализует операцию поиска значения по ключу. При этом ключ преобразуется в число (*хеш-код*), которое используется для быстрого нахождения нужного значения в хеш-таблице.

Таблица 13.1. Пример хеш-таблицы

Ключ	Значение
boy	мальчик
girl	девочка
dog	собачка

Преобразование выполняется с помощью *хеш-функции*, или *функции расстановки*. Эта функция обычно производит какие-либо преобразования внутреннего представления ключа, например, вычисляет среднее арифметическое кодов составляющих его символов (английское слово «hash» означает перемалывать, перемешивать). Если хеш-функция распределяет совокупность возможных ключей равномерно по множеству индексов массива, то доступ к элементу по ключу выполняется почти так же быстро, как в массиве. Если хеш-функция генерирует для разных ключей одинаковые хеш-коды, время поиска возрастает и становится сравнимым со временем поиска в списке.

ПРИМЕЧАНИЕ

Грубый пример хеш-функции: когда мы обращаемся к англо-русскому словарю, мы можем использовать в качестве хеш-кода первую букву английского слова (ключа). Открыв словарь на странице, с которой начинаются слова на эту букву, выполняем последовательный поиск ключа в списке.

Смысл хеш-функции состоит в том, чтоб отобразить более широкое множество ключей в более узкое множество индексов. При этом неизбежно возникают так называемые *коллизии*, когда хеш-функция формирует для двух разных элементов один и тот же хеш-код. В разных реализациях хеш-таблиц используются различные стратегии борьбы с коллизиями.

Граф — это совокупность узлов и ребер, соединяющих различные узлы. Например, можно представить себе карту автомобильных дорог как граф с городами в качестве узлов и шоссе между городами в качестве ребер. Множество реальных практических задач можно описать в терминах графов, что делает их структурой данных, часто используемой при написании программ.

Множество — это неупорядоченная совокупность элементов. Для множеств определены операции проверки принадлежности элемента множеству, включения и исключения элемента, а также объединения, пересечения и вычитания множеств.

Описанные структуры данных называются *абстрактными*, поскольку в них не задается реализация допустимых операций.

В библиотеках большинства современных объектно-ориентированных языков программирования представлены стандартные классы, реализующие основные

абстрактные структуры данных. Такие классы называются *коллекциями*, или *контейнерами*. Для каждого типа коллекции определены методы работы с ее элементами, не зависящие от конкретного типа данных, которые хранятся в коллекции, поэтому один и тот же вид коллекции можно использовать для хранения данных различных типов. Использование коллекций позволяет сократить сроки разработки программ и повысить их надежность.

ВНИМАНИЕ

Каждый вид коллекции поддерживает свой набор операций над данными, и быстродействие этих операций может быть разным. Выбор вида коллекции зависит от того, что требуется делать с данными в программе и какие требования предъявляются к ее быстродействию. Например, при необходимости часто вставлять и удалять элементы из середины последовательности следует использовать список, а не массив, а если включение элементов выполняется главным образом в конец или начало последовательности — очередь. Поэтому изучение возможностей стандартных коллекций и их грамотное применение являются необходимыми условиями создания эффективных и профессиональных программ.

В библиотеке .NET определено множество стандартных классов, реализующих большинство перечисленных ранее абстрактных структур данных. Основные пространства имен, в которых описаны эти классы, — `System.Collections`, `System.Collections.Specialized` и `System.Collections.Generic` (начиная с версии 2.0). В следующих разделах кратко описаны основные элементы этих пространств имен.

ПРИМЕЧАНИЕ

Класс `System.Array`, представляющий базовую функциональность массива, описан в разделе «Класс System.Array» (см. с. 133).

Пространство имен System.Collections

В пространстве имен `System.Collections` определены наборы стандартных коллекций и интерфейсов, которые реализованы в этих коллекциях. В табл. 13.2 приведены наиболее важные интерфейсы, часть из которых мы уже изучали в разделе «Стандартные интерфейсы .NET» (см. с. 198).

Таблица 13.2. Интерфейсы пространства имен System.Collections

Интерфейс	Назначение
`ICollection`	Определяет общие характеристики (например, размер) для набора элементов
`IComparer`	Позволяет сравнивать два объекта
`IDictionary`	Позволяет представлять содержимое объекта в виде пар «имя—значение»
`IDictionaryEnumerator`	Используется для нумерации содержимого объекта, поддерживающего интерфейс `IDictionary`

продолжение

Таблица 13.2 (*продолжение*)

Интерфейс	Назначение
IEnumerable	Возвращает интерфейс IEnumerator для указанного объекта
IEnumerator	Обычно используется для поддержки оператора foreach в отношении объектов
IHashCodeProvider	Возвращает хеш-код для реализации типа с применением выбранного пользователем алгоритма хеширования
IList	Поддерживает методы добавления, удаления и индексирования элементов в списке объектов

В табл. 13.3 перечислены основные коллекции, определенные в пространстве System.Collections[1].

Таблица 13.3. Коллекции пространства имен System.Collections

Класс	Назначение	Важнейшие из реализованных интерфейсов
ArrayList	Массив, динамически изменяющий свой размер	IList, ICollection, IEnumerable, ICloneable
BitArray	Компактный массив для хранения битовых значений	ICollection, IEnumerable, ICloneable
Hashtable	Хеш-таблица[2]	IDictionary, ICollection, IEnumerable, ICloneable
Queue	Очередь	ICollection, ICloneable, IEnumerable
SortedList	Коллекция, отсортированная по ключам. Доступ к элементам — по ключу или по индексу	IDictionary, ICollection, IEnumerable, ICloneable
Stack	Стек	ICollection, IEnumerable

Пространство имен System.Collections.Specialized включает специализированные коллекции, например коллекцию строк StringCollection и хеш-таблицу со строковыми ключами StringDictionary.

В качестве примера стандартной коллекции рассмотрим класс ArrayList.

Класс ArrayList

Основным недостатком обычных массивов является то, что объем памяти, необходимый для хранения их элементов, должен быть выделен до начала работы с массивом. Класс ArrayList позволяет программисту не заботиться о выделении памяти и хранить в одном и том же массиве элементы различных типов.

---

[1] Таблицы 13.2 и 13.3 приводятся по [27] и документации.

[2] У типов, экземпляры которых предназначены для хранения в объекте класса Hashtable, должен быть замещен метод System.Object.GetHashCode.

По умолчанию при создании объекта типа ArrayList строится массив из 16 элементов типа object. Можно задать желаемое количество элементов в массиве, передав его в конструктор или установив в качестве значения свойства Capacity, например:

```
ArrayList arr1 = new ArrayList();      // создается массив из 16   элементов
ArrayList arr2 = new ArrayList(1000);  // создается массив из 1000 элементов
ArrayList arr3 = new ArrayList();
arr3.Capacity = 1000;                        // количество элементов задается
```

Основные методы и свойства класса ArrayList перечислены в табл. 13.4.

Таблица 13.4. Основные элементы класса ArrayList

Элемент	Вид	Описание
Capacity	Свойство	Емкость массива (количество элементов, которые могут храниться в массиве)
Count	Свойство	Фактическое количество элементов массива
Item	Свойство	Получить или установить значение элемента по заданному индексу
Add	Метод	Добавление элемента в конец массива
AddRange	Метод	Добавление серии элементов в конец массива
BinarySearch	Метод	Двоичный поиск в отсортированном массиве или его части
Clear	Метод	Удаление всех элементов из массива
Clone	Метод	Поверхностное копирование[1] элементов одного массива в другой массив
CopyTo	Метод	Копирование всех или части элементов массива в одномерный массив
GetRange	Метод	Получение значений подмножества элементов массива в виде объекта типа ArrayList
IndexOf	Метод	Поиск первого вхождения элемента в массив (возвращает индекс найденного элемента или –1, если элемент не найден)
Insert	Метод	Вставка элемента в заданную позицию (по заданному индексу)
InsertRange	Метод	Вставка группы элементов, начиная с заданной позиции
LastIndexOf	Метод	Поиск последнего вхождения элемента в одномерный массив
Remove	Метод	Удаление первого вхождения заданного элемента в массив
RemoveAt	Метод	Удаление элемента из массива по заданному индексу
RemoveRange	Метод	Удаление группы элементов из массива
Reverse	Метод	Изменение порядка следования элементов на обратный
SetRange	Метод	Установка значений элементов массива в заданном диапазоне
Sort	Метод	Упорядочивание элементов массива или его части
TrimToSize	Метод	Установка емкости массива равной фактическому количеству элементов

[1] Это понятие рассматривалось в разделе «Клонирование объектов» (см. с. 205).

Класс ArrayList реализован через класс Array, то есть содержит закрытое поле этого класса. Поскольку все типы в C# являются потомками класса object, массив может содержать элементы произвольного типа. Даже если в массиве хранятся обычные целые числа, то есть элементы значимого типа, внутренний класс является массивом ссылок на экземпляры типа object, которые представляют собой упакованный тип-значение. Соответственно, при занесении в массив выполняется упаковка, а при извлечении — распаковка элемента. Это не может не сказаться на быстродействии алгоритмов, использующих ArrayList.

Если при добавлении элемента в массив оказывается, что фактическое количество элементов массива превышает его емкость, она автоматически удваивается, то есть происходит повторное выделение памяти и переписывание туда всех существующих элементов.

Пример занесения элементов в экземпляр класса ArrayList:

```
arr1.Add( 123 );
arr1.Add( -2 );
arr1.Add( "Вася" );
```

Доступ к элементу выполняется по индексу, однако при этом необходимо явным образом привести полученную ссылку к целевому типу, например:

```
int    a = (int)    arr1[0];
int    b = (int)    arr1[1];
string s = (string) arr1[2];
```

Попытка приведения к типу, не соответствующему хранимому в элементе, вызывает генерацию исключения InvalidCastException.

Для повышения надежности программ применяется следующий прием: экземпляр класса ArrayList объявляется закрытым полем класса, в котором необходимо хранить коллекцию значений определенного типа, а затем описываются методы работы с этой коллекцией, делегирующие свои функции методам ArrayList. Этот способ иллюстрируется в листинге 13.1, где создается класс для хранения объектов типа Monster и производных от него. По сравнению с аналогичным листингом из раздела «Виртуальные методы» (см. с. 178), в котором используется обычный массив, у нас появилась возможность хранить в классе Stado произвольное количество элементов.

Листинг 13.1. Коллекция объектов

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
    class Monster { ... }
    class Daemon : Monster { ... }

    class Stado : IEnumerable
    {
```

```
        private ArrayList list;
        public Stado()                { list = new ArrayList(); }
        public void Add( Monster m )  { list.Add( m ) ; }
        public void RemoveAt( int i ) { list.RemoveAt( i ); }
        public void Clear()           { list.Clear(); }
        public IEnumerator GetEnumerator()
            { return list.GetEnumerator(); }
    }
    class Class1
    {   static void Main()
        {
            Stado stado = new Stado();

            stado.Add( new Monster( "Monia" ) );
            stado.Add( new Monster( "Monk" ) );
            stado.Add( new Daemon ( "Dimon", 3 ) );

            stado.RemoveAt( 1 );
            foreach ( Monster x in stado ) x.Passport();
        }
    }
}
```

Результат работы программы:

```
Monster Monia     health = 100 ammo = 100
Daemon Dimon      health = 100 ammo = 100 brain = 3
```

Недостатком этого решения является то, что для каждого метода стандартной коллекции приходится описывать метод-оболочку, вызывающий стандартный метод. Хотя это и несложно, но несколько неизящно. В C#, начиная с версии 2.0, появились классы-прототипы (generics), позволяющие решить эту проблему. Мы рассмотрим их в следующем разделе.

Классы-прототипы

Многие алгоритмы не зависят от типов данных, с которыми они работают. Простейшими примерами могут служить сортировка и поиск. Возможность отделить алгоритмы от типов данных предоставляют *классы-прототипы* (generics) — классы, имеющие в качестве параметров типы данных. Чаще всего эти классы применяются для хранения данных, то есть в качестве контейнерных классов, или коллекций.

ПРИМЕЧАНИЕ

Английский термин «generics» переводится в нашей литературе по-разному: универсальные классы, родовые классы, параметризованные классы, обобщенные классы, шаблоны, классы-прототипы и даже просто генерики. Наиболее адекватными мне кажутся варианты «классы-прототипы» и «параметризованные классы», поэтому в последующем изложении в основном используются именно эти термины, хотя и их точными не назовешь.

Во вторую версию библиотеки .NET добавлены *параметризованные коллекции* для представления основных структур данных, применяющихся при создании программ, — стека, очереди, списка, словаря и т. д. Эти коллекции, расположенные в пространстве имен `System.Collections.Generic`, дублируют аналогичные коллекции пространства имен `System.Collections`, рассмотренные в разделе «Пространство имен System.Collections» (см. с. 295). В табл. 13.5 приводится соответствие между обычными и параметризованными коллекциями библиотеки .NET (параметры, определяющие типы данных, хранимых в коллекции, указаны в угловых скобках).

Таблица 13.5. Параметризованные коллекции библиотеки .NET версии 2.0

Класс-прототип (версия 2.0)	Обычный класс
`Comparer<T>`	`Comparer`
`Dictionary<K,T>`	`HashTable`
`LinkedList<T>`	—
`List<T>`	`ArrayList`
`Queue<T>`	`Queue`
`SortedDictionary<K,T>`	`SortedList`
`Stack<T>`	`Stack`

У коллекций, описанных в библиотеке .NET версий 1.0 и 1.1, есть два основных недостатка, обусловленных тем, что в них хранятся ссылки на тип `object`:

- в одной и той же коллекции можно хранить элементы любого типа, следовательно, ошибки при помещении в коллекцию невозможно проконтролировать на этапе компиляции, а при извлечении элемента требуется его явное преобразование;
- при хранении в коллекции элементов значимых типов выполняется большой объем действий по упаковке и распаковке элементов, что в значительной степени снижает эффективность работы.

Параметром класса-прототипа является тип данных, с которым он работает. Это избавляет от перечисленных недостатков. В качестве примера рассмотрим применение универсального «двойника» класса `ArrayList` — класса `List<T>` — для хранения коллекции объектов классов `Monster` и `Daemon`, которые разрабатывались в главах 5 и 8, а также для хранения целых чисел[1].

Листинг 13.2. Использование универсальной коллекции `List<T>`

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

[1] Полное описание этих классов приведено в разделе «Создание библиотеки» (см. с. 275).

```
namespace ConsoleApplication1
{
    using MonsterLib;
    class Program
    {
        static void Main()
        {
            List<Monster> stado = new List<Monster>();
            stado.Add( new Monster( "Monia" ) );
            stado.Add( new Monster( "Monk" ) );
            stado.Add( new Daemon ( "Dimon", 3 ) );

            foreach ( Monster x in stado ) x.Passport();

            List<int> lint = new List<int>();
            lint.Add( 5 ); lint.Add( 1 ); lint.Add( 3 );
            lint.Sort();
            int a = lint[2];
            Console.WriteLine( a );

            foreach ( int x in lint ) Console.Write( x + " " );
        }
    }
}
```

Результат работы программы:

```
Monster Monia    health = 100 ammo = 100
Monster Monk     health = 100 ammo = 100
Daemon Dimon     health = 100 ammo = 100 brain = 3
5
1 3 5
```

В листинге 13.2 две коллекции. Первая (`stado`) содержит элементы пользовательских классов, которые находятся в библиотеке MonsterLib.dll, созданной в предыдущей главе. В коллекции, для которой объявлен тип элементов `Monster`, благодаря полиморфизму можно хранить элементы любого производного класса, но не элементы других типов.

Казалось бы, по сравнению с обычными коллекциями это ограничение, а не универсальность, однако на практике коллекции, в которых действительно требуется хранить значения различных, не связанных межу собой типов, почти не используются. Достоинством же такого ограничения является то, что компилятор может выполнить контроль типов во время компиляции, а не выполнения программы, что повышает ее надежность и упрощает поиск ошибок.

Коллекция `lint` состоит из целых чисел, причем для работы с ними не требуются ни операции упаковки и распаковки, ни явные преобразования типа при получении элемента из коллекции, как это было в обычных коллекциях (см. листинг 13.1).

Классы-прототипы называют также родовыми или шаблонными, поскольку они представляют собой образцы, по которым во время выполнения программы

строятся конкретные классы. При этом сведения о классах, которые являются параметрами классов-прототипов, извлекаются из метаданных.

ВНИМАНИЕ

Использование стандартных параметризованных коллекций для хранения и обработки данных является хорошим стилем программирования, поскольку позволяет сократить сроки разработки программ и повысить их надежность. Рекомендуется тщательно изучить по документации свойства и методы этих классов и выбирать наиболее подходящие в зависимости от решаемой задачи.

В листинге 13.3 приведен еще один пример применения параметризованных коллекций. Программа считывает содержимое текстового файла, разбивает его на слова и подсчитывает количество повторений каждого слова в тексте. Для хранения слов и числа их повторений используется словарь `Dictionary<T,K>`. У этого класса два параметра: тип ключей и тип значений, хранимых в словаре. В качестве ключей используются слова, считанные из файла, а значения представляют собой счетчики целого типа, которые увеличиваются на единицу, когда слово встречается в очередной раз.

Листинг 13.3. Формирование частотного словаря

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
namespace ConsoleApplication1
{
    class Program
    {   static void Main()
      {
          StreamReader f = new StreamReader( @"d:\C#\text.txt" );       // 1
          string s = f.ReadToEnd();                                     // 2

          char[] separators = { '.', ' ', ',', '!' };                   // 3
          List<string> words = new List<string>( s.Split(separators) ); // 4
          Dictionary<string, int> map = new Dictionary<string, int>();  // 5

          foreach ( string w in words )                                 // 6
          {
              if ( map.ContainsKey( w ) ) map[w]++;
              else                        map[w] = 1;
          }

          foreach ( string w in map.Keys )                              // 7
              Console.WriteLine( "{0}\t{1}", w, map[w] );
      }
    }
}
```

Пусть исходный файл text.txt содержит строки

```
Ехал Грека через реку. Видит Грека, в реке рак.
Сунул Грека в реку руку, рак за руку Греку цап!
```

Тогда результат работы программы будет выглядеть так:

```
Ехал    1
Грека   3
через   1
реку    2
        4
Видит   1
в       2
реке    1
рак     2

Сунул   1
руку    2
за      1
Греку   1
цап     1
```

Несколько пояснений к программе. В операторе 1 открывается текстовый файл, длина которого не должна превышать 32 767 символов, потому что в операторе 2 все его содержимое считывается в отдельную строку.

ПРИМЕЧАНИЕ

Конечно, для реальной работы такой способ не рекомендуется. Кроме того, для файлов, открываемых для чтения, программа обязательно должна обрабатывать исключение FileNotFoundException (см. главу 11).

В операторе 3 задается массив разделителей, передаваемый в качестве параметра методу Split, формирующему массив строк, каждая из которых содержит отдельное слово исходного файла. Этот массив используется для инициализации экземпляра words класса List<string>. Применение стандартного класса позволяет не заботиться о выделении места под массив слов.

Оператор 5 описывает словарь, а в цикле 6 выполняется его заполнение путем просмотра списка слов words. Если слово встречается впервые, в значение, соответствующее слову как ключу, заносится единица. Если слово уже встречалось, значение увеличивается на единицу.

В цикле 7 выполняется вывод словаря путем просмотра всех его ключей (для этого используется свойство словаря Keys, возвращающее коллекцию ключей) и выборки соответствующих значений.

Метод Split не очень интеллектуален: он рассматривает пробел, расположенный после знака препинания, как отдельное слово. Для более точного разбиения на слова используются регулярные выражения, которые рассматриваются в главе 15.

ПРИМЕЧАНИЕ

Обратите внимание на то, насколько использование стандартных коллекций сокращает исходный текст программы. Конечно, на тщательное изучение их возможностей требуется много времени, однако это окупается многократно.

Для полноты картины следует добавить, что наряду с параметризованными классами в пространстве имен `System.Collections.Generic` описаны параметризованные интерфейсы, перечисленные в табл. 13.6.

Таблица 13.6. Параметризованные интерфейсы библиотеки .NET версии 2.0

Параметризованный интерфейс (версия 2.0)	Обычный интерфейс
`ICollection<T>`	`ICollection`
`IComparable<T>`	`IComparable`
`IDictionary<K,T>`	`IDictionary`
`IEnumerable<T>`	`IEnumerable`
`IEnumerator<T>`	`IEnumerator`
`IList<T>`	`IList`

Создание класса-прототипа

Язык C# позволяет создавать собственные классы-прототипы и их разновидности — интерфейсы, структуры, делегаты и события, а также обобщенные (generic) методы обычных классов.

Рассмотрим создание класса-прототипа на примере стека, приведенном в спецификации C#. Параметр типа данных, которые хранятся в стеке, указывается в угловых скобках после имени класса, а затем используется таким же образом, как и обычные типы:

```
public class Stack<T>
{
    T[] items;
    int count;
    public void Push( T item ) { ... }      // помещение в стек
    public T    Pop()          { ... }      // извлечение из стека
}
```

При использовании этого класса на место параметра `T` подставляется реальный тип, например `int`:

```
Stack<int> stack = new Stack<int>();
stack.Push( 3 );
int x = stack.Pop();
```

Тип `Stack<int>` называется *сконструированным типом* (constructed type). Этот тип создается во время выполнения программы при его первом упоминании

в программе. Если в программе встретится класс Stack с другим значимым типом, например double, среда выполнения создаст другую копию кода для этого типа. Напротив, для всех ссылочных типов будет использована одна и та же копия кода, поскольку работа с указателями на различные типы выполняется одинаковым образом. Создание конкретного экземпляра класса-прототипа называется *инстанцированием*.

Класс-прототип может содержать произвольное количество параметров типа. Для каждого из них могут быть заданы *ограничения* (constraints), указывающие, каким требованиям должен удовлетворять аргумент, соответствующий этому параметру, например, может быть указано, что это должен быть значимый тип или тип, который реализует некоторый интерфейс.

Синтаксически ограничения задаются после ключевого слова where, например:

```
public class Stack<T>
where T : struct
{ ...
```

Здесь с помощью слова struct записано ограничение, что элементы стека должны быть значимого типа. Для ссылочного типа употребляется ключевое слово class. Для каждого типа, являющегося параметром класса, может быть задана одна строка ограничений, которая может включать один класс, а за ним — произвольное количество интерфейсов, перечисляемых через запятую.

Указание в качестве ограничения имени класса означает, что соответствующий аргумент при инстанцировании может быть именем либо этого класса, либо его потомка. Указание имени интерфейса означает, что тип-аргумент должен реализовывать данный интерфейс — это позволяет использовать внутри класса-прототипа, например, операции по перечислению элементов, их сравнению и т. п.

Помимо класса и интерфейсов в ограничениях можно задать требование, чтобы тип-аргумент имел конструктор по умолчанию без параметров, что позволяет создавать объекты этого типа в теле методов класса-прототипа. Это требование записывается в виде выражения new(), например:

```
public class EntityTable<K,E>
where K: IComparable<K>, IPersistable
where E: Entity, new()
{
    public void Add( K key, E entity )
    {   ...
        if ( key.CompareTo( x ) < 0 ) { ... }
        ...
    }
}
```

В этом примере на первый аргумент класса EntityTable накладываются два ограничения по интерфейсам, а для второго аргумента задано, что он может быть только классом Entity или его потомком и иметь конструктор без параметров.

Задание ограничений позволяет компилятору выполнять более строгий контроль типов и, таким образом, избежать многих ошибок, которые иначе проявились бы только во время выполнения программы.

Для задания значений по умолчанию типизированным элементам класса-прототипа используется ключевое слово `default`. С его помощью элементам ссылочного типа присваивается значение `null`, а элементам значимого типа — 0.

Обобщенные методы

Иногда удобно иметь отдельный метод, параметризованный каким-либо типом данных. Рассмотрим этот случай на примере метода сортировки.

Известно, что «самого лучшего» алгоритма сортировки не существует. Для различных объема, диапазона, степени упорядоченности данных и распределения ключей оптимальными могут оказаться разные алгоритмы. Стандартные методы сортировки реализуют алгоритмы, которые хороши для большинства применений, но не для всех, поэтому может возникнуть необходимость реализовать собственный метод.

В листинге 13.4 приведен пример сортировки методом выбора. Алгоритм состоит в том, что сначала выбирается наименьший элемент массива и меняется местами с первым элементом, затем просматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым элементом, и т. д., всего $n - 1$ раз. На последнем проходе цикла при необходимости меняются местами предпоследний и последний элементы массива.

Листинг 13.4. Сортировка выбором

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApplication1
{   class Program
    {
        static void Sort<T> ( ref T[] a )                                    // 1
            where T : IComparable<T>                                         // 2
        {
            T buf;
            int n = a.Length;
            for ( int i = 0; i < n - 1; ++i )
            {
                int im = i;
                for ( int j = i + 1; j < n; ++j )
                    if ( a[j].CompareTo(a[im]) < 0 ) im = j;                 // 3
                buf = a[i]; a[i] = a[im]; a[im] = buf;
            }
        }
        static void Main()
```

```
        {
            int[] a = { 1, 6, 4, 2, 7, 5, 3 };
            Sort<int>( ref a );                                    // 4
            foreach ( int elem in a ) Console.WriteLine( elem );

            double[] b = { 1.1, 5.2, 5.21, 2, 7, 6, 3 };
            Sort( ref b );                                         // 5
            foreach ( double elem in b ) Console.WriteLine( elem );

            string[] s = { "qwe", "qwer", "df", "asd" };
            Sort( ref s );                                         // 6
            foreach ( string elem in s ) Console.WriteLine( elem );
        }
    }
}
```

Метод параметризован типом, на который накладывается ограничение (оператор 2), чтобы объекты класса-аргумента можно было сравнивать друг с другом с помощью метода `CompareTo`, использованного в операторе 3.

В главной программе (методе `Main`) метод `Sort` вызывается двумя способами: с явным указанием параметра-типа (оператор 4) и без указания параметра (операторы 5 и 6). Во втором случае компилятор по типу переданного в метод параметра самостоятельно определяет, какой именно тип используется при инстанцировании.

ПРИМЕЧАНИЕ

Параметры-типы могут использоваться в списке параметров, возвращаемом типе и в теле универсального метода.

Итак, параметризованные типы и методы позволяют:

- описывать способы хранения и алгоритмы обработки данных независимо от типов данных;
- выполнять контроль типов во время компиляции, а не исполнения программы;
- увеличить скорость обработки данных за счет исключения операций упаковки, распаковки и преобразования типа.

Как уже упоминалось, помимо классов-прототипов и обобщенных методов можно описать параметризованные интерфейсы, структуры и делегаты.

В помощью *параметризованных интерфейсов* можно определить список функций, которые могут быть реализованы различным образом для разных классов, реализующих эти интерфейсы. Параметризованные интерфейсы можно реализовывать в классе-прототипе, используя в качестве аргументов интерфейса параметры типа, реализующего интерфейс, или в обычном классе, подставляя в качестве параметров интерфейса конкретные типы.

Параметризованные делегаты позволяют создавать обобщенные алгоритмы, логику которых можно изменять передаваемыми в качестве параметров делегатами.

Частичные типы

Во вторую версию языка введена возможность разбивать описание типа на части и хранить их в разных физических файлах, создавая так называемые *частичные типы* (partial types). Это может потребоваться для классов большого объема или, что более актуально, для отделения части кода, сгенерированной средствами среды, от написанной программистом вручную. Кроме того, такая возможность облегчает отладку программы, позволяя отделить отлаженные части класса от новых.

Для описания отдельной части типа используется модификатор `partial`. Он может применяться к классам, структурам и интерфейсам, например:

```
public partial class A
{
    ...
}
public partial class A
{
    ...
}
```

После совместной компиляции этих двух частей получается такой же код, как если бы класс был описан обычным образом. Все части одного и того же частичного типа должны компилироваться одновременно, иными словами, добавление новой части к уже скомпилированным не допускается.

Модификатор `partial` не является ключевым словом и должен стоять непосредственно перед одним из ключевых слов `class`, `struct` или `interface` в каждой из частей. Все части определения одного класса должны быть описаны в одном и том же пространстве имен.

ПРИМЕЧАНИЕ

Если модификатор `partial` указывается для типа, описание которого состоит только из одной части, это не является ошибкой.

Модификаторы доступа для всех частей типа должны быть согласованными. Если хотя бы одна из частей содержит модификатор `abstract` или `sealed`, класс считается соответственно абстрактным или бесплодным.

Класс-прототип также может объявляться по частям, в этом случае во всех частях должны присутствовать одни и те же параметры типа с одними и теми же ограничениями.

Если частичный тип является наследником нескольких интерфейсов, в каждой части не требуется перечислять все интерфейсы: обычно в одной части объявляется один интерфейс и описывается его реализация, в другой части — другой интерфейс и т. д. Набором базовых интерфейсов для типа, объявленного в нескольких частях, является объединение базовых интерфейсов, определенных в каждой части.

Обнуляемые типы

В программировании давно существует проблема, каким образом задать, что переменной не присвоено никакого значения. Эта проблема решается разными способами. Один из способов заключается в том, чтобы присвоить переменной какое-либо значение, не входящее в диапазон допустимых для нее. Например, если величина может принимать только положительные значения, ей присваивается –1. Ясно, что для многих случаев этот подход неприменим. Другой способ — хранение логического признака, по которому можно определить, присвоено ли переменной значение. Этот способ не может использоваться, например, для значений, возвращаемых из метода.

В версии C# 2.0 указанная проблема решается введением типов специального вида, называемых *обнуляемыми* (nullable). Обнуляемый тип представляет собой структуру, хранящую наряду со значением величины (свойство `Value`) логический признак, по которому можно определить, было ли присвоено значение этой величине (свойство `HasValue`).

Если значение величине было присвоено, свойство `HasValue` имеет значение `true`. Если значение величины равно `null`, свойство `HasValue` имеет значение `false`, а попытка получить значение через свойство `Value` вызывает генерацию исключения.

Обнуляемый тип строится на основе базового типа, за которым следует символ ?, например:

```
int? x = 123;
int? y = null;
if ( x.HasValue ) Console.WriteLine( x ); // вместо x можно записать x.Value
if ( y.HasValue ) Console.WriteLine( y );
```

Существуют явные и неявные *преобразования* из обнуляемых типов в обычные и обратно, при этом выполняется контроль возможности получения значения, например:

```
int     i = 123;
int?    x = i;              // int     --> int?
double? y = x;              // int?    --> double?
int?    z = (int?) y;       // double? --> int?
int     j = (int) z;        // int?    --> int
```

Для величин обнуляемых типов определены *операции отношения* . Операции == и != возвращают значение `true`, если обе величины имеют значение `null`. Естественно, что значение `null` считается не равным любому ненулевому значению. Операции <, >, <= и >= дают в результате `false`, если хотя бы один из операндов имеет значение `null`.

Арифметические операции с величинами обнуляемых типов дают в результате `null`, если хотя бы один из операндов равен `null`, например:

```
int? x = null;
int? y = x + 1;          // y = null
```

Для величин обнуляемых типов введена еще одна операция — *объединения* ?? (null coalescing operator). Это бинарная операция, результат которой равен первому операнду, если он не равен null, и второму в противном случае. Иными словами, эта операция предоставляет замещаемое значение для null, например:

```
int? x = null;
int  y = x ?? 0;          // y = 0
x = 1;
y = x ?? 0;               // y = 1
```

Обнуляемые типы удобно использовать при работе с базами данных и XML.

Рекомендации по программированию

Алгоритм работы программы во многом зависит от способа организации ее данных, поэтому очень важно до начала разработки алгоритма выбрать оптимальные структуры данных, основываясь на требованиях к функциональности и быстродействию программы.

Для разных задач необходимы различные способы хранения и обработки данных, поэтому необходимо хорошо представлять себе как характеристики и области применения абстрактных структур данных, так и их конкретную реализацию в виде коллекций библиотеки. Изучение возможностей стандартных коллекций и их грамотное применение является необходимым условием создания эффективных и профессиональных программ, позволяет сократить сроки разработки программ и повысить их надежность.

Недостатками коллекций первых версий библиотеки .NET является отсутствие контроля типов на этапе компиляции и неэффективность при хранении элементов значимых типов. *Параметризованные коллекции*, появившиеся в версии 2.0 библиотеки, избавлены от этих недостатков, поэтому в программах рекомендуется использовать именно коллекции версии 2.0, выбирая наиболее подходящие классы в зависимости от решаемой задачи.

Для реализации алгоритмов, независимых от типов данных, следует использовать *классы-прототипы* и *обобщенные методы*. Они не снижают эффективность программы по сравнению с обычными классами и методами, поскольку код для конкретного типа генерируется средой CLR во время выполнения программы. Помимо классов-прототипов и обобщенных методов можно описать параметризованные интерфейсы, структуры и делегаты.

Частичные типы удобно использовать при разработке объемных классов группой программистов и для упрощения отладки программ. *Обнуляемые типы* применяют для работы с данными, для которых необходимо уметь определять, было ли им присвоено значение.

Глава 14

Введение в программирование под Windows

В предыдущих главах мы изучали возможности языка C# на примере консольных приложений, чтобы не распылять свое внимание на изучение классов библиотеки .NET, поддерживающих стандартный графический интерфейс пользователя. Сейчас пришло время освоить принципы создания Windows-приложений и получить представление об основных классах библиотеки, которые при этом используются.

В первую очередь рассмотрим *основные особенности Windows* как характерного и широко распространенного представителя современных операционных систем.

- *Многозадачность* — это возможность одновременно выполнять несколько приложений. Операционная система обеспечивает разделение ресурсов: каждому приложению выделяется свое адресное пространство, распределяется процессорное время и организуются очереди для доступа к внешним устройствам. Внутри приложения также можно организовать параллельное выполнение нескольких фрагментов, называемых потоками. В разных версиях Windows используются различные механизмы диспетчеризации.
- *Независимость программ от аппаратуры.* Для управления аппаратными средствами любое приложение обращается к операционной системе, что обеспечивает независимость от конкретных физических характеристик устройств: при смене устройства никакие изменения в программу вносить не требуется. Управление внешними устройствами обеспечивается с помощью драйверов.
- *Стандартный графический интерфейс с пользователем.* Основное взаимодействие с пользователем осуществляется в графическом режиме. Каждое приложение выполняет вывод в отведенную ему прямоугольную область экрана, называемую *окном.* Окно состоит из стандартных элементов. Это упрощает освоение программ пользователем и облегчает работу программиста, поскольку в его распоряжение предоставляются библиотеки интерфейсных компонентов. Такие библиотеки входят в состав систем программирования.

Интерфейсные компоненты обращаются к аппаратуре не непосредственно, а через функции операционной системы, называемые API (Application Program Interface — программный интерфейс приложения). API-функции находятся в динамических библиотеках (Dynamic Link Library, DLL), разделяемых всеми приложениями. Эти библиотеки называются динамическими потому, что находящиеся в них функции не подключаются к каждому исполняемому файлу до выполнения программы, а вызываются в момент обращения к ним.

В основе пользовательского интерфейса Windows лежит представление экрана как *рабочего стола*, на котором располагаются «листы бумаги» — окна приложений, ярлыки и меню. Одни элементы могут полностью или частично перекрывать другие, пользователь может изменять их размеры и перемещать их. Приложение может иметь несколько окон, одно из них является главным. При закрытии главного окна приложение завершается.

- *Поддержка виртуального адресного пространства для каждого приложения.* Каждому приложению доступно пространство адресов оперативной памяти размером до 4 Гбайт[1]. Операционная система отображает его на физические адреса и обеспечивает защиту приложений друг от друга. В разных версиях Windows защита выполняется с различной степенью надежности, например, системы Windows 95/98 гораздо менее надежны, чем Windows NT/2000.
- *Возможность обмена данными между приложениями.* Приложения могут обмениваться данными через буфер обмена или используя другие механизмы, например OLE (Object Linking and Embedding — связывание и внедрение объектов) или именованные программные каналы.
- *Возможность запуска старых программ.* В 32-разрядных версиях Windows можно выполнять 16-разрядные Windows-программы, а также программы, написанные под MS-DOS. Последние запускаются в так называемой виртуальной DOS-машине, которая создает у программы полное «впечатление» того, что она выполняется под управлением MS-DOS в монопольном режиме.
- *Принцип событийного управления* (рассматривается в следующем разделе).

Событийно-управляемое программирование

В основу Windows положен принцип *событийного управления*. Это значит, что и сама система, и приложения после запуска ожидают действий пользователя и реагируют на них заранее заданным образом. Любое действие пользователя (нажатие клавиши на клавиатуре, щелчок кнопкой мыши, перемещение мыши) называется *событием*. Структура программы, управляемой событиями, изображена на рис. 14.1.

---

[1] Имеются в виду 32-разрядные приложения, то есть приложения, в которых используется адрес длиной 32 разряда.

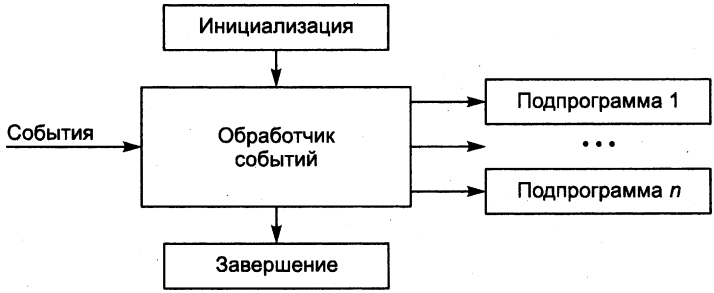


Рис. 14.1. Структура программы, управляемой событиями

Событие воспринимается Windows и преобразуется в *сообщение* — запись, содержащую необходимую информацию о событии (например, какая клавиша была нажата, в каком месте экрана произошел щелчок мышью). Сообщения могут поступать не только от пользователя, но и от самой системы, а также от активного или других приложений. Определен достаточно широкий круг стандартных сообщений, образующий иерархию, кроме того, можно определять собственные сообщения.

Сообщения поступают в общую очередь, откуда распределяются по очередям приложений. Каждое приложение содержит *цикл обработки сообщений*, который выбирает сообщение из очереди и через операционную систему вызывает подпрограмму, предназначенную для его обработки (рис. 14.2). Таким образом, Windows-приложение состоит из главной программы, содержащей цикл обработки сообщений, инициализацию и завершение приложения, и набора *обработчиков событий*.

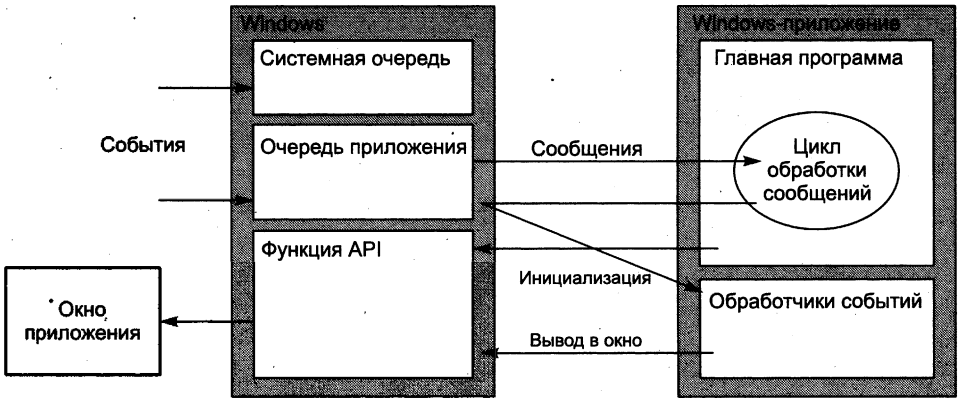


Рис. 14.2. Структура Windows-приложения

Среда Visual Studio.NET содержит удобные средства разработки Windows-приложений, выполняющие вместо программиста рутинную работу — создание шаблонов приложения и форм, заготовок обработчиков событий, организацию цикла обработки сообщений и т. д. Рассмотрим эти средства.

Шаблон Windows-приложения

Создадим новый проект (File ▸ New ▸ Project), выбрав шаблон Windows Application (рис. 14.3). После более длительных раздумий, чем для консольного приложения, среда сформирует шаблон Windows-приложения. Первое отличие, которое бросается в глаза, — вкладка *заготовки формы* Form1.cs[Design], расположенная в основной части экрана. Форма представляет собой окно и предназначена для размещения *компонентов* (элементов управления) — меню, текста, кнопок, списков, изображений и т. д.

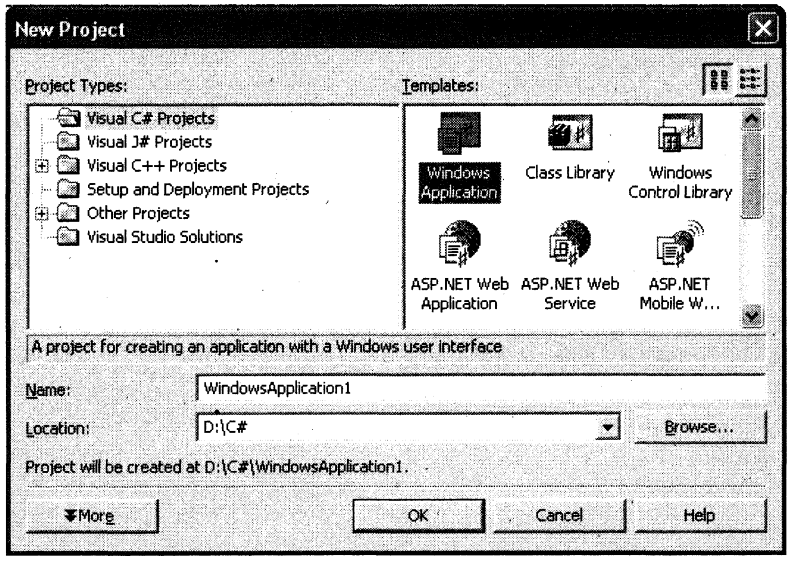


Рис. 14.3. Выбор шаблона проекта

Среда создает не только заготовку формы, но и шаблон текста приложения. Перейти к нему можно, щелкнув в окне Solution Explorer (View ▸ Solution Explorer) правой кнопкой мыши на файле Form1.cs и выбрав в контекстном меню команду View Code. При этом откроется вкладка с кодом формы, который, за исключением комментариев, приведен в листинге 14.1. Представлять себе, что написано в вашей программе, весьма полезно, поэтому давайте внимательно рассмотрим этот текст.

Листинг 14.1. Шаблон Windows-приложения

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

```
namespace WindowsApplication1
{
    public class Form1 : System.Windows.Forms.Form
    {
        private System.ComponentModel.Container components = null;

        public Form1()
        {
            InitializeComponent();
        }

        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        #region Windows Form Designer generated code
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.Size = new System.Drawing.Size(300,300);
            this.Text = "Form1";
        }
        #endregion

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

Приложение начинается с директив использования пространств имен библиотеки .NET. Для пустой формы, не содержащей ни одного компонента, необходимыми являются только две директивы:

```
using System;
using System.Windows.Forms;
```

Остальные директивы добавлены средой «на вырост». С пространством имен `System` вы уже знакомы. Пространство `System.Windows.Forms` содержит огромное количество типов, являющихся строительными блоками Windows-приложений.

Список наиболее употребительных элементов этого пространства имен приведен в табл. 14.1, а часть иерархии — на рис. 14.4.

Таблица 14.1. Основные типы Windows.Forms

Класс	Назначение
`Application`	Класс Windows-приложения. При помощи методов этого класса можно обрабатывать Windows-сообщения, запускать и прекращать работу приложения и т. п.
`ButtonBase`, `Button`, `CheckBox`, `ComboBox`, `DataGrid`, `GroupBox`, `ListBox`, `LinkLabel`, `PictureBox`	Примеры классов, представляющих элементы управления (компоненты): базовый класс кнопок, кнопка, флажок, комбинированный список, таблица, группа, список, метка с гиперссылкой, изображение
`Form`	Класс формы — окно Windows-приложения
`ColorDialog`, `FileDialog`, `FontDialog`, `PrintPreviewDialog`	Примеры стандартных диалоговых окон для выбора цветов, файлов, шрифтов, окно предварительного просмотра
`Menu`, `MainMenu`, `MenuItem`, `ContextMenu`	Классы выпадающих и контекстных меню
`Clipboard`, `Help`, `Timer`, `Screen`, `ToolTip`, `Cursors`	Вспомогательные типы для организации графических интерфейсов: буфер обмена, помощь, таймер, экран, подсказка, указатели мыши
`StatusBar`, `Splitter`, `ToolBar`, `ScrollBar`	Примеры дополнительных элементов управления, размещаемых на форме: строка состояния, разделитель, панель инструментов и т. д.

Мы будем изучать типы пространства имен `Windows.Forms` по мере необходимости.

Продолжим рассмотрение листинга 14.1. В нем описан класс `Form1`, производный от класса `Form`. Он наследует от своего предка множество элементов, которые мы рассмотрим в следующих разделах. В самом классе `Form1` описано новое закрытое поле `components` — контейнер для хранения компонентов, которые можно добавить в класс формы.

Конструктор формы вызывает закрытый метод `InitializeComponent`, автоматически формируемый средой (код метода скрыт между директивами препроцессора `#region` и `#endregion`). Этот метод обновляется средой при добавлении элементов управления на форму, а также при изменении свойств формы и содержащихся на ней элементов. Например, если изменить цвет фона формы с помощью окна свойств (Properties), в методе появится примерно такая строка:

```
this.BackColor = System.Drawing.SystemColors.AppWorkspace;
```

Метод освобождения ресурсов `Dispose` вызывается автоматически при закрытии формы. Точка входа в приложение, метод `Main`, содержит вызов статического метода `Run` класса `Application`. Метод `Run` запускает цикл обработки сообщений и выводит на экран форму, новый экземпляр которой передан ему в качестве параметра.

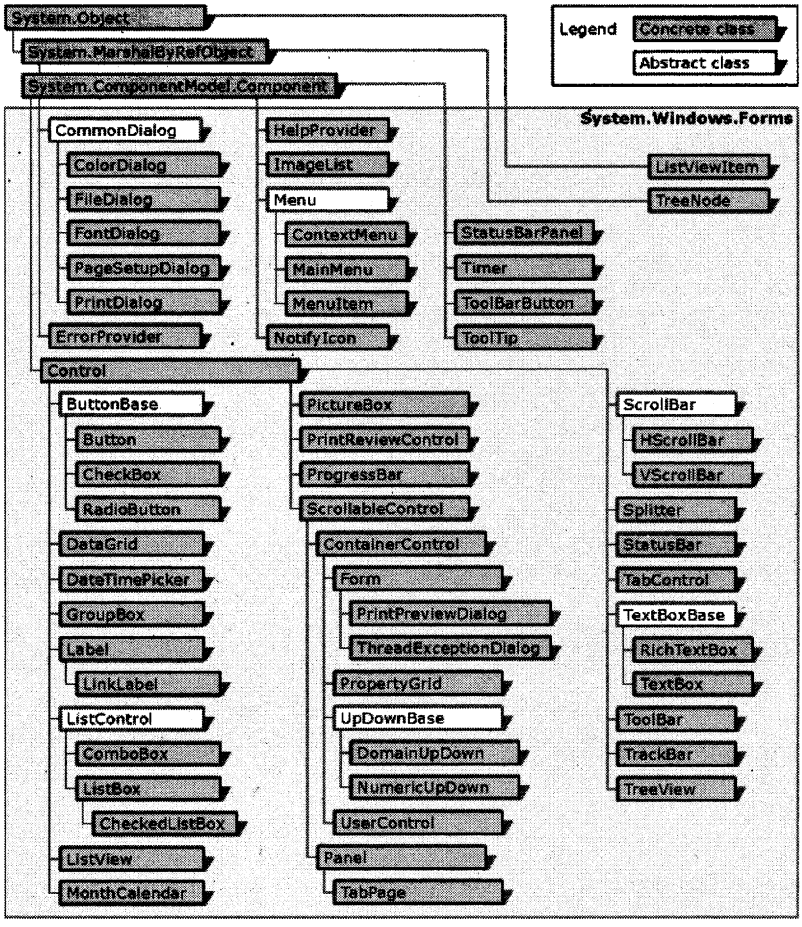


Рис. 14.4. Элементы управления Windows.Forms

ПРИМЕЧАНИЕ

Запуск приложения, для создания которого мы пока не написали ни строчки кода, можно выполнить с помощью команды меню Debug ▸ Start или клавиши F5. На экран выводится пустая форма, обладающая стандартной функциональностью окна Windows-приложения: например, она умеет изменять свои размеры и реагировать на действия с кнопками развертывания и закрытия.

Процесс создания Windows-приложения состоит из двух основных этапов:

1. *Визуальное проектирование*, то есть задание внешнего облика приложения.
2. *Определение поведения* приложения путем написания процедур обработки событий.

Визуальное проектирование заключается в помещении на форму компонентов (элементов управления) и задании их свойств и свойств самой формы с помощью окна свойств. Если его не видно, можно воспользоваться командой меню

View ▸ Properties Window. Свойства отображаются либо в алфавитном порядке, либо сгруппированными по категориям. Способ отображения выбирается с помощью кнопок, расположенных в верхней части окна свойств (рис. 14.5).

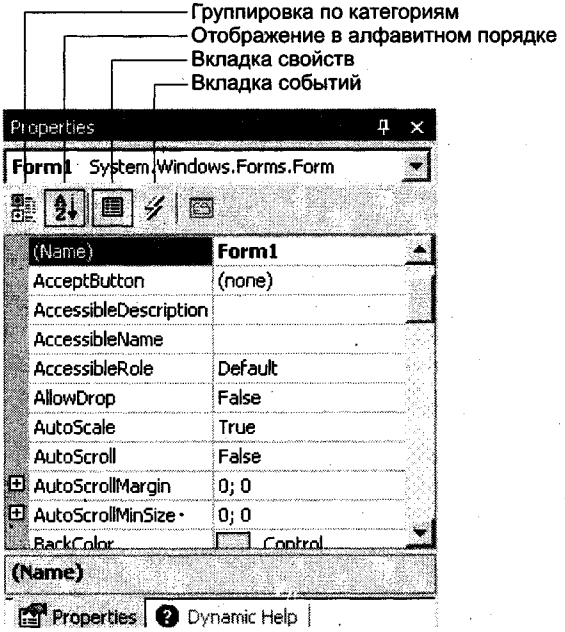


Рис. 14.5. Окно свойств

Самый простой способ размещения компонента — двойной щелчок на соответствующем значке палитры компонентов Toolbox (если ее не видно, можно воспользоваться командой меню View ▸ Toolbox), при этом компонент помещается на форму. Затем компонент можно переместить и изменить его размеры с помощью мыши. Можно также сделать один щелчок на палитре и еще один щелчок в том месте формы, где планируется разместить компонент. В окне свойств отображаются свойства выбранного в данный момент компонента (он окружен рамкой).

Задание свойств выполняется либо выбором имеющихся в списке вариантов, либо вводом требуемого значения с клавиатуры. Если около имени свойства стоит значок +, это означает, что свойство содержит другие свойства. Они становятся доступными после щелчка на значке. Пример формы, на которой размещены компоненты с вкладки палитры Windows Forms, показан на рис. 14.6. Эти компоненты описаны в разделе «Элементы управления» (см. с. 325).

Определение поведения программы начинается с принятия решений, какие действия должны выполняться при щелчке на кнопках, вводе текста, выборе пунктов меню и т. д., иными словами, по каким событиям будут выполняться действия, реализующие функциональность программы.

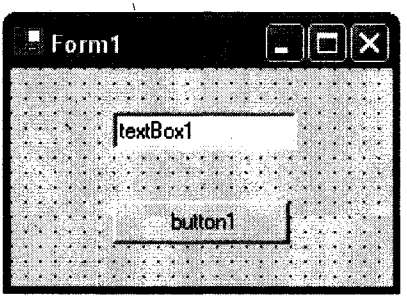


Рис. 14.6. Форма с размещенными на ней компонентами

ВНИМАНИЕ

Интерфейс программы должен быть интуитивно понятным и по возможности простым. Часто повторяющиеся действия не должны требовать от пользователя выполнения сложных последовательностей операций.

Заготовка шаблона обработчика события формируется двойным щелчком на поле, расположенном справа от имени соответствующего события на вкладке Events окна свойств, при этом появляется вкладка окна редактора кода с заготовкой соответствующего обработчика (вкладка Events отображается в окне свойств при щелчке на кнопке с желтой молнией, как показано на рис. 14.5).

Для каждого класса определен свой набор событий, на которые он может реагировать. Наиболее *часто используемые события*:

- `Activated` — получение формой фокуса ввода;
- `Click`, `DoubleClick` — одинарный и двойной щелчки мышью;
- `Closed` — закрытие формы;
- `Load` — загрузка формы;
- `KeyDown`, `KeyUp` — нажатие и отпускание любой клавиши и их сочетаний;
- `KeyPress` — нажатие клавиши, имеющей ASCII-код;
- `MouseDown`, `MouseUp` — нажатие и отпускание кнопки мыши;
- `MouseMove` — перемещение мыши;
- `Paint` — возникает при необходимости прорисовки формы.

В листинге 14.2 приведен текст программы, сформированной средой после размещения на форме компонентов, представленных на рис. 14.6, и выбора для кнопки события `Click`, а для поля ввода — события `KeyPress`. Из листинга для удобства восприятия удалены лишние элементы.

Листинг 14.2. Шаблон приложения с двумя компонентами и заготовками обработчиков событий

```
using System;
using System.Drawing;
using System.Collections;
```

продолжение

Листинг 14.2 (*продолжение*)

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

namespace WindowsApplication1
{
    public class Form1 : System.Windows.Forms.Form                    // 0
    {
        private System.Windows.Forms.TextBox textBox1;                // 1
        private System.Windows.Forms.Button  button1;                 // 2
        private System.ComponentModel.Container components = null;

        public Form1()
        {
            InitializeComponent();
        }

        protected override void Dispose( bool disposing ) { ... }

        #region Windows Form Designer generated code
        private void InitializeComponent()
        {
            this.textBox1 = new System.Windows.Forms.TextBox();       // 3
            this.button1  = new System.Windows.Forms.Button();        // 4
            this.SuspendLayout();                                     // 5
            //
            // textBox1
            //
            this.textBox1.Location = new System.Drawing.Point(24, 16);
            this.textBox1.Name = "textBox1";
            this.textBox1.Size = new System.Drawing.Size(240, 20);
            this.textBox1.TabIndex = 0;
            this.textBox1.Text = "textBox1";
            this.textBox1.KeyPress += new                             // 6
          System.Windows.Forms.KeyPressEventHandler(this.textBox1_KeyPress);
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(192, 80);
            this.button1.Name = "button1";
            this.button1.TabIndex = 1;
            this.button1.Text = "button1";
            this.button1.Click += new                                 // 7
                 System.EventHandler(this.button1_Click);
            //
            // Form1
            //
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(292, 126);
            this.Controls.Add(this.button1);                          // 8
```

```
            this.Controls.Add(this.textBox1);                          // 9
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);                                  // 10
        }
        #endregion

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

        private void button1_Click(object sender, System.EventArgs e)
        {
                                                                       // 11
        }

        private void textBox1_KeyPress(object sender,
                                System.Windows.Forms.KeyPressEventArgs e)
        {
                                                                       // 12
        }
    }
}
```

Рассмотрим текст программы. В операторах 1 и 2 в класс формы добавляются два закрытых поля: строка ввода типа TextBox и кнопка типа Button. Эти типы определены в пространства имен System.Windows.Forms, которое подключается в соответствующей директиве using, поэтому можно записать эти и соседние строки проще:

```
public class Form1 : Form                                              // 0
{
    private TextBox   textBox1;                                        // 1
    private Button    button1;                                         // 2
    private Container components = null;
```

Чаще всего программист изменяет имена, присваиваемые элементам средой, более осмысленными. Это можно сделать, задав новое значение свойства Name элемента.

ПРИМЕЧАНИЕ

Того же самого эффекта можно достичь и с помощью вкладки Class View (View ▸ Class View), выбрав в списке нужный элемент и изменив значение его свойства Name в окне свойств. Обратите внимание на то, что при этом в нем отображаются свойства кнопки button1 не как элемента интерфейса, а как поля класса. Можно изменить имена и вручную, но это более трудоемко и чревато ошибками.

Самое важное происходит в методе InitializeComponent.

В операторах 3 и 4 создаются экземпляры компонентов, затем для каждого из них задаются свойства, определяющие их положение, размер, вид и т. д. Обратите

внимание на операторы 6 и 7. В них регистрируются обработчики соответствующих событий. Механизм обработки событий тот же, что мы рассматривали в главе 10 (см. с. 232) — он описывается моделью «публикация — подписка».

Например, для кнопки button1, в составе которой есть событие Click, регистрируется обработчик button1_Click, являющийся закрытым методом класса Form1. Это значит, что при наступлении события нажатия на кнопку (об этом сообщит операционная система) будет вызван соответствующий обработчик.

Имя обработчика формируется средой автоматически из имени экземпляра компонента и имени события. Обратите внимание на то, что обработчикам передаются два параметра: объект-источник события и запись, соответствующая типу события.

ПРИМЕЧАНИЕ

При задании обработчика можно задать и другое имя, для этого оно записывается справа от имени соответствующего события на вкладке Events окна свойств.

После создания экземпляров компонентов и настройки их свойств компоненты заносятся в коллекцию, доступ к которой выполняется через свойство Controls (операторы 8 и 9). Если этого не сделать, компоненты не будут отображаться на форме. Коллекция поддерживает методы добавления и удаления компонентов (Add и Remove).

Таким образом, для *размещения компонента на форме* необходимо выполнить три действия:

1. *Создать экземпляр* соответствующего класса.
2. *Настроить свойства экземпляра*, в том числе зарегистрировать обработчик событий.
3. *Поместить экземпляр в коллекцию* компонентов формы.

Операторы 5 и 10 используются для того, чтобы все изменения в свойства элементов вносились одновременно. Для этого в операторе 5 внесение изменений «замораживается», а в операторе 10 разрешается.

В теле обработчиков событий (операторы 11 и 12) программист может наконец-то самостоятельно написать код, который будет выполняться при наступлении события. Добавим в эти строки операторы, выводящие *окна сообщений* с соответствующим текстом:

```
MessageBox.Show("Нажата кнопка button1");                    // 11
MessageBox.Show("Нажата клавиша " + e.KeyChar);              // 12
```

Здесь используется простейший вариант статического метода Show класса MessageBox, определенного в пространстве имен System.Windows.Forms. Существуют более десяти перегруженных вариантов этого метода, позволяющих варьировать вид выводимой информации, например задать заголовок окна и наименования отображаемых на нем кнопок.

Прежде чем приступить к изучению форм и элементов управления, размещаемых на формах, необходимо рассмотреть их общего предка — класс Control.

Класс Control

Класс Control является базовым для всех отображаемых элементов, то есть элементов, которые составляют графический интерфейс пользователя, например кнопок, списков, полей ввода и форм. Класс Control реализует базовую функциональность интерфейсных элементов. Он содержит методы обработки ввода пользователя с помощью мыши и клавиатуры, определяет размер, положение, цвет фона и другие характеристики элемента. Для каждого объекта можно определить родительский класс, задав свойство Parent, при этом объект будет иметь, например, такой же цвет фона, как и его родитель[1].

Наиболее важные свойства класса Control перечислены в табл. 14.2, методы — в табл. 14.3.

Таблица 14.2. Основные свойства класса Control

Свойство	Описание
Anchor	Определяет, какие края элемента управления будут привязаны к краям родительского контейнера. Если задать привязку всех краев, элемент будет изменять размеры вместе с родительским
BackColor, BackgroundImage, Font, ForeColor, Cursor	Определяют параметры отображения рабочей области формы: цвет фона, фоновый рисунок, шрифт, цвет текста, вид указателя мыши
Bottom, Right	Координаты нижнего правого угла элемента. Могут устанавливаться также через свойство Size
Top, Left	Координаты верхнего левого угла элемента. Эквивалентны свойству Location
Bounds	Возвращает объект типа Rectangle (прямоугольник), который определяет размеры элемента управления
ClientRectangle	Возвращает объект Rectangle, определяющий размеры рабочей области элемента
ContextMenu	Определяет, какое контекстное меню будет выводиться при щелчке на элементе правой кнопкой мыши
Dock	Определяет, у какого края родительского контейнера будет отображаться элемент управления
Location	Координаты верхнего левого угла элемента относительно верхнего левого угла контейнера, содержащего этот элемент, в виде структуры типа Point. Структура содержит свойства X и Y
Height, Width	Высота и ширина элемента
Size	Высота и ширина элемента в виде структуры типа Size. Структура содержит свойства Height и Width

продолжение

[1] Речь идет не о наследовании, а о взаимоотношениях объектов во время выполнения программы. Например, если на форме размещена кнопка, форма является родительским объектом по отношению к кнопке.

Таблица 14.2 (*продолжение*)

Свойство	Описание
`Created`, `Disposed`, `Enabled`, `Focused`, `Visible`	Возвращают значения типа `bool`, определяющие текущее состояние элемента: создан, удален, использование разрешено, имеет фокус ввода, видимый
`Handle`	Возвращает дескриптор элемента (уникальное целочисленное значение, сопоставленное элементу)
`ModifierKeys`	Статическое свойство, используемое для проверки состояния модифицирующих клавиш (**Shift**, **Control**, **Alt**). Возвращает результат в виде объекта типа `Keys`
`MouseButtons`	Статическое свойство, проверяющее состояние клавиш мыши. Возвращает результат в виде объекта типа `MouseButtons`
`Opacity`	Определяет степень прозрачности элемента управления. Может изменяться от 0 (прозрачный) до 1 (непрозрачный)
`Parent`	Возвращает объект, родительский по отношению к данному (имеется в виду не базовый класс, а объект-владелец)
`Region`	Определяет объект `Region`, при помощи которого можно управлять очертаниями и границами элемента управления
`TabIndex`, `TabStop`	Используются для настройки последовательности перемещения с помощью клавиши **Tab** по элементам управления, расположенным на форме

Таблица 14.3. Основные методы класса Control

Метод	Описание
`Focus`	Установка фокуса ввода на элемент[1]
`GetStyle`, `SetStyle`	Получение и установка флагов управления стилем элемента. Используются значения перечисления `ControlStyles` (см. далее)
`Hide`, `Show`	Управление свойством `Visible` (`Hide` — скрыть элемент, `Show` — отобразить элемент)
`Invalidate`	Обновление изображения элемента путем отправки соответствующего сообщения в очередь сообщений. Метод перегружен таким образом, чтобы можно было обновлять не всю область, занимаемую элементом, а лишь ее часть
`On`*XXXX*	Методы-обработчики событий (`OnMouseMove`, `OnKeyDown`, `OnResize`, `OnPaint` и т. п.), которые могут быть замещены в производных классах
`Refresh`	Обновление элемента и всех его дочерних элементов
`SetBounds`, `SetLocation`, `SetClientArea`	Управление размером и положением элемента

[1] В элемент, имеющий *фокус ввода*, направляется ввод пользователя с клавиатуры.

Перечисление ControlStyles задает возможные значения стиля формы в виде битовых флагов, поэтому можно использовать их комбинации. Значения всех констант перечисления можно посмотреть в электронной документации, а для первого знакомства достаточно одного — ResizeRedraw. Этот стиль определяет, что при изменении размеров формы она будет автоматически перерисована. По умолчанию перерисовка не выполняется, и если на форме есть какое-либо изображение, результат изменения размеров формы может сильно озадачить.

В табл. 14.4 перечислена небольшая часть событий, определенных в классе Control.

Таблица 14.4. Некоторые события класса Control

Событие	Описание
Click, DoubleClick, MouseEnter, MouseLeave, MouseDown, MouseUp, MouseMove, MouseWheel	События от мыши
KeyPress, KeyUp, KeyDown	События от клавиатуры
BackColorChanged, ContextMenuChanged, FontChanged, Move, Paint, Resize	События изменения элемента
GotFocus, Leave, LostFocus	События получения и потери фокуса ввода

Применение наиболее важных элементов, описанных в таблицах, рассматривается в следующих разделах.

При написании приложений применяются два способа обработки событий:

- замещение стандартного обработчика;
- задание собственного обработчика.

В большинстве случаев применяется второй способ. Среда разработки создает заготовку обработчика по двойному щелчку на поле, расположенном справа от имени соответствующего события на вкладке Events окна свойств. При этом в код приложения автоматически добавляется строка, регистрирующая этот обработчик.

Первый способ, то есть переопределение виртуальных методов On*XXXX* (OnMouseMove, OnKeyDown, OnResize, OnPaint и т. п.), применяется в основном тогда, когда перед реакцией на событие требуется выполнить какие-либо дополнительные действия. За подробностями интересующиеся могут обратиться к [27].

Элементы управления

Элементы управления, или *компоненты*, помещают на форму с помощью панели инструментов ToolBox (View ▸ ToolBox). В этом разделе кратко описаны простейшие компоненты и приведены примеры их использования.

Обычно компоненты применяют в диалоговых окнах для получения данных от пользователя или его информирования.

Метка Label

Метка предназначена для размещения текста на форме. Текст хранится в свойстве Text. Можно задавать шрифт (свойство Font), цвет фона (BackColor), цвет шрифта (ForeColor) и выравнивание (TextAlign) текста метки. Метка может автоматически изменять размер в зависимости от длины текста (AutoSize = True). Можно разместить на метке изображение (Image) и задать прозрачность (установить для свойства BackColor значение Color.Transparent).

Метка не может получить фокус ввода, следовательно, не обрабатывает сообщения от клавиатуры. Пример использования меток приведен далее (см. с. 327).

Кнопка Button

Основное событие, обрабатываемое кнопкой, — щелчок мышью (Click). Кроме того, кнопка может реагировать на множество других событий — нажатие клавиш на клавиатуре и мыши, изменение параметров и т. д. Нажатие клавиши Enter или пробела, если при этом кнопка имеет фокус ввода, эквивалентно щелчку мышью на кнопке.

Можно изменить начертание и размер шрифта текста кнопки, который хранится в свойстве Text, задать цвет фона и фоновое изображение так же, как и для метки.

Если занести имя кнопки в свойство AcceptButton формы, на которой расположена кнопка, то нажатие клавиши Enter вызывает событие Click, даже если кнопка не имеет фокуса ввода. Такая кнопка имеет дополнительную рамку и называется *кнопкой по умолчанию*.

Аналогично, если занести имя кнопки в свойство CancelButton формы, на которой расположена кнопка, то нажатие клавиши Esc вызывает событие Click для этой кнопки.

Кнопки часто используются в *диалоговых окнах*. Как видно из названия, такое окно предназначено для диалога с пользователем и запрашивает у него какие-либо сведения (например, какой выбрать режим или какой файл открыть). Диалоговое окно обладает свойством *модальности*. Это означает, что дальнейшие действия с приложением невозможны до того момента, пока это окно не будет закрыто. Закрыть окно можно, либо подтвердив введенную в него информацию щелчком на кнопке OK (или Yes), либо отменив ее с помощью кнопки закрытия окна или, например, кнопки Cancel.

Таблица 14.5. Значения перечисления DialogResult

Значение	Описание	Значение	Описание
None	Окно не закрывается	Ignore	Нажата кнопка Ignore
OK	Нажата кнопка OK	Yes	Нажата кнопка Yes
Cancel	Нажата кнопка Cancel	No	Нажата кнопка No
Abort	Нажата кнопка Abort	Retry	Нажата кнопка Retry

Для сохранения информации о том, как было закрыто окно, у кнопки определяют свойство DialogResult. Это свойство может принимать стандартные значения из перечисления DialogResult, определенного в пространстве имен System.Windows.Forms. Значения перечисления приведены в табл. 14.5.

Пример использования кнопок приведен в следующем разделе.

Поле ввода TextBox

Компонент TextBox позволяет пользователю вводить и редактировать текст, который запоминается в свойстве Text. Можно вводить строки практически неограниченной длины (приблизительно до 32 000 символов), корректировать их, а также вводить защищенный текст (пароль) путем установки маски, отображаемой вместо вводимых символов (свойство PasswordChar). Для обеспечения возможности ввода нескольких строк устанавливают свойства Multiline, ScrollBars и WordWrap. Доступ только для чтения устанавливается с помощью свойства ReadOnly.

Элемент содержит методы очистки (Clear), выделения (Select), копирования в буфер (Copy), вставки из него (Paste) и др., а также реагирует на множество событий, основными из которых являются KeyPress и KeyDown.

Рассмотрим создание простого приложения, в котором использованы компоненты типа Label, Button и TextBox. Главное окно приложения показано на рис. 14.7.

Рис. 14.7. Окно приложения «Диагностика кармы»

Пользователь вводит число и нажимает клавишу Enter, после чего введенное значение сравнивается с числом, «задуманным» генератором случайных чисел. Если пользователь не угадал, выводится сообщение «Не угадали!». Каждое следующее сообщение для наглядности немного сдвигается вправо относительно предыдущего. После совпадения выводится итоговое сообщение, и фокус ввода передается на кнопку Еще раз (рис. 14.8). «Коэффициент невезучести» определяется как количество попыток, деленное на максимально возможное значение числа.

Создание приложения начинается с проектирования его интерфейса. На форме располагаются метки, поля ввода и кнопка. В окне свойств их свойству Text задаются перечисленные в табл. 14.6 значения (в окне отображаются свойства выделенного элемента; элемент выделяется щелчком мыши).

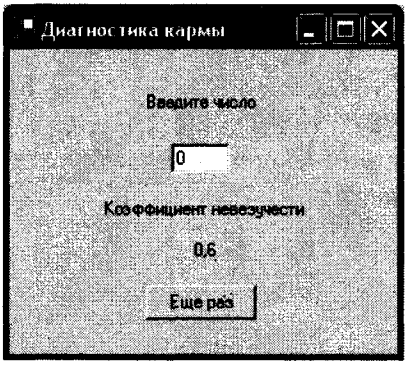


Рис. 14.8. Окно приложения после ввода правильного ответа

Таблица 14.6. Значение свойства Text для компонентов приложения

Компонент	Значение свойства Text
Form1	Диагностика кармы
label1	Введите число
label2, label3, textBox1	Пустое значение[1]
button1	Еще раз

Поведение приложения задается в обработчиках событий. Число надо загадывать до того, как пользователь введет первый ответ, а затем после каждого щелчка на кнопке **Еще раз**. Следовательно, эти действия должны выполняться при загрузке формы (событие `Load`) и щелчке на кнопке (событие `Click`). Ответ пользователя анализируется при нажатии клавиши **Enter** после ввода числа (событие `KeyPress` элемента `textBox1`).

Вызов редактора программ выполняется двойным щелчком рядом с соответствующим событием элемента на вкладке **Events** окна свойств (см. рис. 14.5). Для хранения загаданного числа и ответа пользователя применяются поля `i` и `k`. В константе `max` хранится максимально возможное число. Переменная `rnd` представляет собой экземпляр генератора случайных чисел[2].

Сокращенный текст программы с комментариями приведен в листинге 14.3.

Листинг 14.3. Программа «Угадай число»

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

[1] Необходимо стереть значение, заданное по умолчанию.

[2] Класс `Random` описан в одноименном разделе (см. с. 148).

```
namespace WindowsApplication1
{
    public class Form1 : Form
    {
        private Label   label1;
        private TextBox textBox1;
        private Button  button1;
        private Label   label2;
        private Label   label3;
        private Container components = null;

        public Form1() { ... }
        protected override void Dispose( bool disposing ) { ... }
        Windows Form Designer generated code { ... }

        const int max = 10;     // максимальное значение загадываемого числа
        Random rnd;                  // экземпляр генератора случайных чисел
        int i, k;                                        // загадка и отгадка

        static void Main() { ... }

        private void Form1_Load(object sender, EventArgs e)
        {
            rnd = new Random();
            i = rnd.Next(max);     // загадано число в диапазоне от 0 до max
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
        {
            int n;
            if ( e.KeyChar != (char)13 )
                 return;              // если нажата не клавиша Enter, выйти

            try                 // преобразование ввода пользователя в число
            {
                n = Convert.ToInt32(textBox1.Text);
            }
            catch
            {
                n = -1;  // если введено не число, принять за неверный ответ
            }

            if ( n != i )             // ============ пользователь не угадал
            {
                label2.Left += 5;
                label2.Text = "Не угадали!";
                textBox1.Clear();
                k++;                            // увеличение счетчика попыток
                button1.Visible = false;
            }
```

продолжение ⇩

Листинг 14.3 (*продолжение*)

```
            else                        // ============= пользователь угадал
            {
                label2.Left = 32;  // восстановить начальное положение метки
                label2.Text = "Коэффициент невезучести";
                double koef = 1.0 * k / max;
                label3.Text = koef.ToString();
                button1.Visible = true;
            }
        }

        private void button1_Click(object sender, System.EventArgs e)
        {
            i = rnd.Next(max);      // загадано число в диапазоне от 0 до max
            k = 0;                     // обнуление количества попыток ответа
            textBox1.Clear();                          // поле ввода очищено
            textBox1.Focus();              // курсор установлен на поле ввода
            label2.Text = "";                          // метки очищены
            label3.Text = "";
        }
    }
}
```

Меню MainMenu и ContextMenu

Главное меню `MainMenu` размещается на форме таким же образом, как и другие компоненты: двойным щелчком на его значке на панели Toolbox. При этом значок располагается под заготовкой формы, а среда переходит в режим редактирования пунктов меню. Каждый пункт меню представляет собой объект типа `MenuItem`, и при вводе пункта меню мы задаем его свойство `Text`. Переход к заданию заголовка следующего пункта меню выполняется либо щелчком мыши, либо нажатием клавиши Enter и клавиш со стрелками. Обычно, чтобы сделать программу понятнее, изменяют также свойства `Name` каждого пункта так, чтобы они соответствовали названиям пунктов.

Пункт меню может быть запрещен или разрешен (свойство `Enabled`), видим или невидим (`Visible`), отмечен или не отмечен (`Checked`). Заготовка обработчика событий формируется двойным щелчком на пункте меню.

Любое приложение обычно содержит в меню команду `Exit`, при выборе которой приложение завершается. Для закрытия приложения можно воспользоваться либо методом `Close` класса главной формы приложения, либо методом `Exit` класса `Application`, например:

```
private void Exit_Click(object sender, EventArgs e)
{                                                  // имя пункта меню - Exit
    Close();
    // или:
    // Application.Exit();
}
```

Контекстное меню — это меню, которые вызывается во время выполнения программы по нажатию правой кнопки мыши на форме или элементе управления. Обычно в этом меню размещаются пункты, дублирующие пункты главного меню, или пункты, определяющие специфические для данного компонента действия.

Контекстное меню ContextMenu создается и используется аналогично главному (значок контекстного меню появляется на панели инструментов, если воспользоваться кнопкой прокрутки). Для привязки контекстного меню к компоненту следует установить значение свойства ContextMenu этого компонента равным имени контекстного меню.

Флажок CheckBox

Флажок используется для включения-выключения пользователем какого-либо режима. Для проверки, установлен ли флажок, анализируют его свойство Checked, принимающее значение true или false. Флажок может иметь и третье состояние — «установлен, но не полностью». Как правило, это происходит в тех случаях, когда устанавливаемый режим определяется несколькими «подрежимами», часть из которых включена, а часть выключена. В этом случае используют свойство CheckState, которое может принимать значения Checked, Unchecked и Intermediate.

Кроме того, флажок обладает свойством ThreeState, которое управляет возможностью установки третьего состояния пользователем с помощью мыши. Для флажка можно задать цвет фона и фоновое изображение так же, как и для метки. Свойство Appearance управляет отображением флажка: либо в виде собственно флажка (Normal), либо в виде кнопки (Button), которая «залипает» при щелчке на ней мышью.

Флажки используются в диалоговых окнах как поодиночке, так и в группе, причем все флажки устанавливаются независимо друг от друга. Пример приложения с флажками приведен далее в этой главе.

Переключатель RadioButton

Переключатель позволяет пользователю выбрать один из нескольких предложенных вариантов, поэтому переключатели обычно объединяют в группы. Если один из них устанавливается (свойство Checked), остальные автоматически сбрасываются. Программист может менять стиль и цвет текста, связанного с переключателем, и его выравнивание. Для переключателя можно задать цвет фона и фоновое изображение так же, как и для метки.

Переключатели можно поместить непосредственно на форму, в этом случае все они составят одну группу. Если на форме требуется отобразить несколько групп переключателей, их размещают внутри компонента Group или Panel.

Свойство Appearance управляет отображением переключателя: либо в традиционном виде (Normal), либо в виде кнопки (Button), которая «залипает» при щелчке на ней мышью[1].

Пример использования переключателей приведен далее в этой главе.

[1] При установке для свойства AutoCheck значения false кнопка не «залипает».

Панель GroupBox

Панель GroupBox служит для группировки элементов на форме, например для того, чтобы дать общее название и визуально выделить несколько переключателей или флажков, обеспечивающих выбор связанных между собой режимов.

Приведенная в листинге 14.4 программа запрашивает у пользователя, массив какой длины он хочет создать, и создает целочисленный массив с помощью генератора случайных чисел. Пользователь может выбрать диапазон значений элементов: либо [–10; 10], либо [–100; 100]. После создания массива можно вычислить его максимальный элемент и/или количество положительных элементов. Окно приложения показано на рис. 14.9. Для удобства имена (свойство Name) большинства компонентов изменены, как указано на рисунке. Переключатели размещены на панели типа GroupBox. Установлено свойство Checked компонента radioButton1, очищены все поля ввода, а для полей maxtextBox и numPosittextBox установлено свойство ReadOnly (только для чтения).

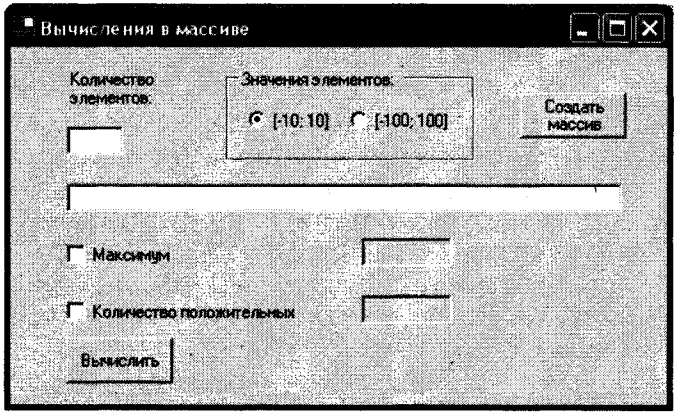


Рис. 14.9. Окно приложения для вычислений в массиве

ПРИМЕЧАНИЕ

При визуальном проектировании важно размещать компоненты приятным глазу образом. При этом удобно пользоваться командами меню Format, предварительно выделив нужные компоненты, «обводя» их мышью или выделяя мышью при нажатой клавише Shift или Ctrl. Например, можно выровнять компоненты по левому краю, выделив их и воспользовавшись командой меню Format ▸ Align ▸ Lefts.

Все действия в программе выполняются щелчками на двух кнопках, то есть обрабатываются два события Click.

Листинг 14.4. Вычисления в массиве

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
```

```
using System.Windows.Forms;
using System.Data;

namespace WindowsApplication1
{
    public class Form1 : Form
    {
        private Label       label1;
        private GroupBox    groupBox1;
        private RadioButton radioButton1;
        private RadioButton radioButton2;
        private TextBox     numtextBox;
        private CheckBox    numPositcheckBox;
        private Button      createbutton;
        private Button      calcbutton;
        private TextBox     maxtextBox;
        private TextBox     numPosittextBox;
        private TextBox     arraytextBox;
        private CheckBox    maxcheckBox;
        private Container   components = null;

        public Form1() { ... }
        protected override void Dispose( bool disposing ) { ... }
        Windows Form Designer generated code { ... }

        int[] arr;                                          // описание массива

        static void Main() { ... }

        private void createbutton_Click(object sender, EventArgs e)
        {
            Random rnd = new Random();
            int a = -10, b =  10;              // диапазон значений элементов
            if ( radioButton2.Checked )
            {
                a = -100; b = 100;                 // корректировка диапазона
            }

            int n = 0;
            try
            {
                n = int.Parse(numtextBox.Text);              // длина массива
            }
            catch
            {
                MessageBox.Show("Введите количество элементов!");
                numtextBox.Clear();
                numtextBox.Focus();
            }
```

продолжение ⇩

Листинг 14.4 (*продолжение*)

```
            arraytextBox.Clear();                    // очистка полей ввода
            maxtextBox.Clear();
            numPosittextBox.Clear();

            if ( n < 0 ) n = -n;          // если введено отрицательное число
            arr = new int[n];                          // создание массива
            for ( int i = 0; i < n; ++i )
            {
                arr[i] = rnd.Next(a, b);          // задание элемента массива
                arraytextBox.Text += "  " + arr[i];          // вывод массива
            }
        }

        private void calcbutton_Click(object sender, EventArgs e)
        {
            int max = arr[0];
            int numPosit = 0;
            for ( int i = 0; i < arr.Length; ++i )
            {
                if ( arr[i] > max ) max = arr[i];          // поиск максимума
                if ( arr[i] > 0 )   ++numPosit;  // количество положительных
            }
            if ( maxcheckBox.Checked )
                maxtextBox.Text = max.ToString();
            else maxtextBox.Text = "";
            if ( numPositcheckBox.Checked )
                numPosittextBox.Text = numPosit.ToString();
            else numPosittextBox.Text = "";
        }
    }
}
```

Список ListBox

Список служит для представления перечней элементов, в которых пользователь может выбрать одно (свойство SelectionMode равно One) или несколько значений (свойство SelectionMode равно MultiSimple или MultiExtended). Если значение свойства SelectionMode равно MultiSimple, щелчок мышью на элементе выделяет его или снимает выделение. Значение MultiExtended позволяет использовать при выделении диапазона строк клавишу **Shift**, а при добавлении элемента — клавишу **Ctrl**, аналогично проводнику Windows. Запретить выделение можно, установив значение свойства SelectionMode, равное None.

Чаще всего используются списки строк, но можно выводить и произвольные изображения. Список может состоять из нескольких столбцов (свойство MultiColumn) и быть отсортированным в алфавитном порядке (Sorted = True).

Элементы списка нумеруются с нуля. Они хранятся в свойстве Items, представляющем собой коллекцию. В Items можно добавлять элементы с помощью методов

Add, AddRange и Insert. Для удаления элементов служат методы Remove и RemoveAt, удаляющие заданный элемент и элемент по заданному индексу соответственно.

Выделенные элементы можно получить с помощью свойств SelectedItems и SelectedIndices, предоставляющих доступ к коллекциям выделенных элементов и их индексов.

В листинге 14.5 приведен пример приложения, которое отображает в списке типа ListBox строки, считанные из входного файла, а затем по щелчку на кнопке Запись выводит выделенные пользователем строки в выходной файл. Вид окна приложения приведен на рис. 14.10.

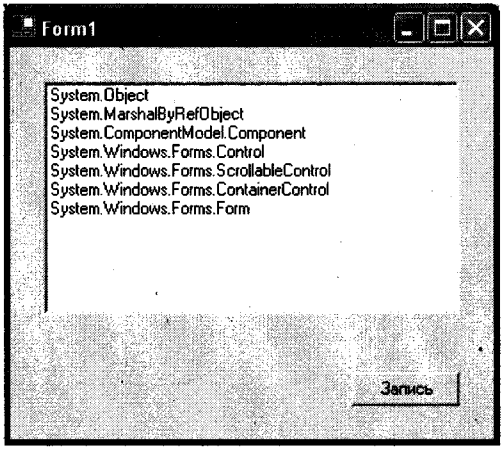


Рис. 14.10. Окно приложения для работы со списком строк

Листинг 14.5. Работа со списком строк

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

namespace WindowsApplication1
{
    public class Form1 : Form
    {
        private ListBox   listBox1;
        private Button    button1;
        private Container components = null;

        public Form1() { ... }
        protected override void Dispose( bool disposing ) { ... }
        Windows Form Designer generated code { ... }
```

продолжение

Листинг 14.5 (*продолжение*)

```
        static void Main() { ... }

        private void Form1_Load(object sender, EventArgs e)
        {
            try
            {
                StreamReader f = new StreamReader( "input.txt" );
                string buf;

                while ( ( buf = f.ReadLine() ) != null )  // чтение из файла
                    listBox1.Items.Add(buf);              // занесение в список
            }

            catch ( FileNotFoundException exc )
            {
                MessageBox.Show( exc.Message );
                return;
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            StreamWriter f = new StreamWriter( "output.txt" );

            foreach ( string item in listBox1.SelectedItems )
                f.WriteLine(item);                        // запись в файл
            f.Close();
        }
    }
}
```

На панели инструментов расположено множество компонентов, не рассмотренных в этой книге. Для их изучения следует обратиться к справочной системе .NET. Она организована понятно и единообразно, и даже те, кто не изучал английский язык, смогут при некотором навыке извлечь оттуда необходимую информацию[1].

СОВЕТ

При изучении компонента рекомендуется следующая последовательность действий. Разместите компонент на форме, выделите его, нажмите клавишу F1 и перейдите по ссылке ...overview (обзор). Изучите разделы Remarks и Example, затем перейдите по ссылке ...Members (элементы класса), расположенной в верхней части окна. Попытайтесь получить представление о возможностях изучаемого класса, выделив главные из его свойств и открытых методов. После этого можно вернуться к заготовке приложения и начать экспериментировать со свойствами, а затем — с методами класса.

В следующей части этой главы будут коротко описаны основные элементы классов Form и Application, входящих в состав любого приложения.

[1] Но лучше, конечно, хотя бы немного знать английский язык, это полезно само по себе.

Предварительные замечания о формах

Класс Form наследует от длинной цепочки своих предков множество элементов, определяющих вид и поведение окон различного типа. Генеалогическое древо класса Form выглядит так: Object→MarshalByRefObject→Component→Control→ ScrollableControl→ContainerControl.

Окна приложения могут иметь различные вид и назначение. Все окна можно разделить на модальные и немодальные. *Модальное окно* не позволяет пользователю переключаться на другие окна того же приложения, пока не будет завершена работа с текущим окном[1]. Как уже отмечалось, в виде модальных обычно оформляют *диалоговые окна*, требующие от пользователя ввода какой-либо информации. Модальное окно можно закрыть щелчком на кнопке наподобие OK, подтверждающей введенную информацию, на кнопке закрытия окна или на кнопке вроде Cancel, отменяющей ввод пользователя. Примером модального окна может служить окно сообщений MessageBox, упоминавшееся в разделе «Шаблон Windows-приложения» (см. с. 322).

Немодальное окно позволяет переключаться на другие окна того же приложения. Немодальные окна являются, как правило, информационными. Они используются в тех случаях, когда пользователю желательно предоставить свободу выбора — оставлять на экране какую-либо информацию или нет.

Каждое приложение содержит одно *главное окно*. Класс главного окна приложения содержит точку входа в приложение (статический метод Main). При закрытии главного окна приложение завершается.

В случае использования *многодокументного интерфейса* (Multiple Document Interface, MDI) одно *родительское окно* может содержать другие окна, называемые *дочерними*. При закрытии родительского окна дочерние окна закрываются автоматически. Вид окна определяет его функциональность, например, окно с одинарной рамкой не может изменять свои размеры.

Рассмотрим наиболее интересных предков класса формы. Их элементы наследуются не только формой, но и другими компонентами, такими как поля ввода или кнопки.

Класс MarshalByRefObject наделяет своих потомков некой особенностью, благодаря которой обращение к ним выполняется по ссылке, то есть локальная копия объекта не создается.

Класс Component обеспечивает потомков способностью взаимодействовать с контейнером, в котором они расположены. Кроме того, в нем определен метод Dispose, который автоматически вызывается, когда экземпляр класса более не используется. Поэтому для освобождения ресурсов, связанных с приложением, обычно переопределяют этот метод.

---

[1] Особый вид модального окна — *системное модальное окно* — не позволяет переключаться даже на окна других приложений.

Класс Control, являющийся предком всех интерфейсных элементов, рассмотрен в этой главе ранее. В классе ScrollableControl определены элементы, позволяющие компоненту иметь горизонтальную и вертикальную полосы прокрутки. Свойства AutoScroll и AutoScrollMinSize обеспечивают автоматическое появление полос прокрутки в тех случаях, когда выводимая информация не помещается в компоненте.

Класс ContainerControl обеспечивает своих потомков возможностью управлять размещенными внутри них дочерними компонентами. Например, на форме обычно располагаются несколько кнопок, меток и т. п., а на панели — несколько флажков или переключателей. Свойства и методы класса позволяют установить фокус ввода на элемент или получать информацию о том, какой элемент имеет фокус ввода, а также управлять порядком получения фокуса с помощью свойств TabStop и TabIndex.

Класс Form

Класс Form представляет собой заготовку формы, от которой наследуются классы форм приложения. Помимо множества унаследованных элементов, в этом классе определено большое количество собственных элементов, наиболее употребительные из которых приведены в табл. 14.7–14.9.

Таблица 14.7. Некоторые свойства класса Form

Свойство	Описание
AcceptButton	Позволяет задать кнопку или получить информацию о кнопке, которая будет активизирована при нажатии пользователем клавиши Enter
ActiveMDIChild, IsMDIChild, IsMDIContainer	Свойства предназначены для использования в приложениях с многодокументным интерфейсом (MDI)
AutoScale	Позволяет установить или получить значение, определяющее, будет ли форма автоматически изменять свои размеры, чтобы соответствовать высоте шрифта, используемого на форме, или размерам размещенных на ней компонентов
FormBorderStyle	Позволяет установить или получить стиль рамки вокруг формы (используются значения перечисления FormBorderStyle)
CancelButton	Позволяет задать кнопку или получить информацию о кнопке, которая будет активизирована при нажатии пользователем клавиши Esc
ControlBox	Позволяет установить или получить значение, определяющее, будет ли присутствовать стандартная кнопка системного меню в верхнем левом углу заголовка формы
Menu, MergedMenu	Используются для установки или получения информации о меню на форме
MaximizeBox, MinimizedBox	Определяют, будут ли на форме присутствовать стандартные кнопки восстановления и свертывания в правом верхнем углу заголовка формы

Свойство	Описание
ShowInTaskbar	Определяет, будет ли форма отображаться на панели задач Windows
StartPosition	Позволяет получить или установить значение, определяющее исходное положение формы в момент выполнения программы (используются значения перечисления FormStartPosition). Например, для отображения формы в центре экрана устанавливается значение CenterScreen
WindowState	Определяет состояние отображаемой формы при запуске (используются значения перечисления FormWindowState)

Таблица 14.8. Методы класса Form

Метод	Описание
Activate	Активизирует форму и помещает в нее фокус ввода
Close	Закрывает форму
CenterToScreen	Помещает форму в центр экрана
LayoutMDI	Размещает все дочерние формы на родительской в соответствии со значениями перечисления LayoutMDI
OnResize	Может быть замещен для реагирования на событие Resize
Show	Отображает форму (унаследовано от Control)
ShowDialog	Отображает форму как диалоговое окно (подробнее о диалоговых окнах рассказывается в следующем разделе)

Таблица 14.9. Некоторые события класса Form

Событие	Описание
Activate	Происходит при активизации формы (когда она выходит в активном приложении на передний план)
Closed, Closing	Происходят во время закрытия формы
MDIChildActive	Возникает при активизации дочернего окна

Об использовании некоторых элементов, перечисленных в таблицах, рассказывается в следующем разделе.

Диалоговые окна

В библиотеке .NET нет специального класса для представления диалоговых окон. Вместо этого устанавливаются определенные значения свойств в классе обычной формы. В диалоговом окне можно располагать те же элементы управления, что и на обычной форме. *Диалоговое окно характеризуется*:

- неизменяемыми размерами (FormBorderStyle = FixedDialog);
- отсутствием кнопок восстановления и свертывания в правом верхнем углу заголовка формы (MaximizeBox = False, MinimizedBox = False);

- наличием кнопок наподобие OK, подтверждающей введенную информацию, и Cancel, отменяющей ввод пользователя, при нажатии которых окно закрывается (`AcceptButton = имя_кнопки_OK`, `CancelButton = имя_кнопки_Cancel`);
- установленным значением свойства `DialogResult` для кнопок, при нажатии которых окно закрывается.

Для отображения диалогового окна используется метод `ShowDialog`, который формирует результат выполнения из значений перечисления `DialogResult`, описанных в табл. 14.5. Если пользователь закрыл диалоговое окно щелчком на кнопке наподобие OK, введенную им информацию можно использовать в дальнейшей работе. Закрытие окна щелчком на кнопке вроде Cancel отменяет все введенные данные. Диалоговое окно обычно появляется при выборе пользователем некоторой команды меню на главной форме.

Приведем пример простого приложения, содержащего две формы. На главной форме (рис. 14.11) расположено меню из двух команд — Dialog и Exit. При выборе команды Dialog появляется диалоговое окно, включающее метку Введите информацию, поле ввода текста и две кнопки, OK и Cancel (рис. 14.12).

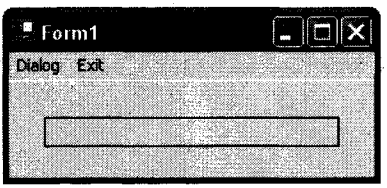


Рис. 14.11. Главное окно приложения с меню и диалоговым окном

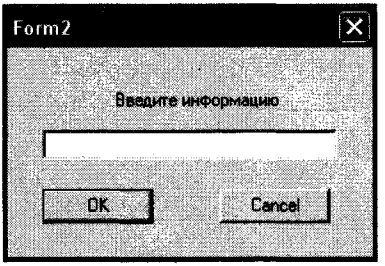


Рис. 14.12. Диалоговое окно

Если пользователь введет текст в диалоговое окно и закроет его щелчком на кнопке OK, этот текст отобразится в поле метки в главном окне. Если диалоговое окно будет закрыто щелчком на кнопке Cancel, поле метки останется неизменным.

Добавление в проект второй формы выполняется выбором в меню команды Project ▸ Add Windows Form. Начнем визуальное проектирование с этой формы. Значения свойств компонентов, установленных с помощью окна свойств, перечислены в табл. 14.10.

Таблица 14.10. Значения свойств элементов формы

Элемент	Свойство	Значение
Метка	Text	Введите информацию
	TextAlign	MiddleCenter
Поле ввода	Text	Пустое значение
Кнопка	Name	btnOK
	DialogResult	OK
	Text	OK
Кнопка	Name	btnCancel
	DialogResult	Cancel
	Text	Cancel
Форма	AcceptButton	btnOK
	CancelButton	btnCancel
	FormBorderStyle	FixedDialog
	MaximizeBox	False
	MinimizeBox	False
	StartPosition	CenterParent

Пользователь вводит информацию в поле ввода textBox1, которое является закрытым элементом формы. Чтобы получить значение свойства Text поля ввода, добавим в описание класса свойство Info. Это единственное дополнение, которое вносится в шаблон текста класса формы:

```
public class Form2 : Form
{
    private Label     label1;
    private TextBox   textBox1;
    private Button    btnOK;
    private Button    btnCancel;
    private Container components = null;
    public string Info
    {
        get
        {
            return textBox1.Text;
        }
    }
    ...
}
```

Визуальное проектирование главной формы еще проще, чем первой, и сводится к размещению главного меню и метки. Для наглядности метка окружена

рамкой (BorderStyle = FixedSingle). Вот как выглядят обработчики событий для пунктов меню:

```
private void menuItem1_Click( object sender, EventArgs e )
{
    Form2 f = new Form2();                  // создание экземпляра класса окна
    if ( f.ShowDialog() == DialogResult.OK )          // отображение окна
        label1.Text = f.Info;
}

private void menuItem2_Click( object sender, EventArgs e )
{
    Close();                                          // закрытие главного окна
}
```

Как видите, *для отображения диалогового окна следует создать экземпляр объекта соответствующей формы, а затем вызвать для этого объекта метод* ShowDialog.

При подтверждении ввода текст пользователя можно получить с помощью свойства Info, доступного только для чтения. При необходимости передавать информацию не только из диалогового окна, но и в него, можно добавить в описание свойства часть set.

ПРИМЕЧАНИЕ

Следует различать процесс создания формы — объекта класса Form или его наследника — от процесса вывода формы на экран. Форма, как и любой объект, создается при выполнении операции new с вызовом конструктора. Для вывода формы служит метод Show или ShowDialog класса Form, вызываемый соответствующим объектом. Для скрытия формы используется метод Hide. Фактически, методы Show и Hide изменяют свойство Visible объекта. Разница между скрытием формы методом Hide и ее закрытием методом Close состоит в том, что первый из них делает форму невидимой, но не изменяет сам объект, а метод Close делает объект недоступным и закрывает все его ресурсы.

Класс Application

Класс Application, описанный в пространстве имен System.Windows.Forms, содержит статические свойства, методы и события, предназначенные для управления приложением в целом и получения его общих характеристик. Наиболее важные элементы класса Application перечислены в табл. 14.11.

Таблица 14.11. Основные элементы класса Application

Элемент класса	Тип	Описание
AddMessageFilter, RemoveMessageFilter	Методы	Позволяют перехватывать сообщения и выполнять с этими сообщениями нужные предварительные действия. Для того чтобы добавить фильтр сообщений, необходимо указать класс, реализующий интерфейс IMessageFilter[1]

[1] Подробности см. в [27].

Элемент класса	Тип	Описание
DoEvents	Метод	Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной операции
Exit	Метод	Завершает работу приложения
ExitThread	Метод	Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток
Run	Метод	Запускает стандартный цикл обработки сообщений для текущего потока
CommonAppDataRegistry	Свойство	Возвращает параметр системного реестра, который хранит общую для всех пользователей информацию о приложении
CompanyName	Свойство	Возвращает имя компании
CurrentCulture	Свойство	Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток
CurrentInputLanguage	Свойство	Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком
ProductName	Свойство	Позволяет получить имя программного продукта, которое ассоциировано с данным приложением
ProductVersion	Свойство	Позволяет получить номер версии программного продукта
StartupPath	Свойство	Позволяет определить имя выполняемого файла для работающего приложения и путь к нему в операционной системе
ApplicationExit	Событие	Возникает при завершении приложения
Idle	Событие	Возникает, когда все текущие сообщения в очереди обработаны и приложение переходит в режим бездействия
ThreadExit	Событие	Возникает при завершении работы потока в приложении. Если работу завершает главный поток приложения, это событие возникает до события ApplicationExit

Многие свойства класса Application позволяют получить метаданные сборки (например, номер версии или имя компании), не используя типы пространства имен System.Reflection. Программист не часто работает непосредственно с классом Application, поскольку большую часть необходимого кода среда формирует автоматически.

Краткое введение в графику

Для вывода линий, геометрических фигур, текста и изображений необходимо создать экземпляр класса Graphics, описанного в пространстве имен System.Drawing. Существуют различные способы создания этого объекта.

Первый способ состоит в том, что ссылку на объект Graphics получают из параметра PaintEventArgs, передаваемого в обработчик события Paint, возникающего при необходимости прорисовки формы или элемента управления:

```
private void Form1_Paint( object sender, PaintEventArgs e )
{   Graphics g = e.Graphics;
    // использование объекта
}
```

Второй способ — использование метода CreateGraphics, описанного в классах формы и элемента управления:

```
Graphics g;
g = this.CreateGraphics();
```

Третий способ — создание объекта с помощью объекта-потомка Image. Этот способ используется для изменения существующего изображения:

```
Bitmap bm = new Bitmap( "d:\\picture.bmp" );
Graphics g = Graphics.FromImage( bm );
```

После создания объекта типа Graphics его можно применять для вывода линий, геометрических фигур, текста и изображений. Основными объектами, которые при этом используются, являются объекты классов:

- Pen — рисование линий и контуров геометрических фигур;
- Brush — заполнение областей;
- Font — вывод текста;
- Color — цвет.

В листинге 14.6 представлен код приложения, в котором на форму выводятся линия, эллипс и текст. Вид формы приведен на рис. 14.13.

Листинг 14.6. Работа с графическими объектами

```
using System;
using System.Drawing;
using System.Windows.Forms;
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1() { InitializeComponent(); }

        private void Form1_Paint( object sender, PaintEventArgs e )
        {
            using ( Graphics g = e.Graphics )                    // 1
            {
```

```
            using ( Pen pen = new Pen( Color.Red ) )                    // 2
            {
                g.DrawLine( pen, 0, 0, 200, 100 );
                g.DrawEllipse( pen, new Rectangle(50, 50, 100, 150) );
            }
                string s = "Sample Text";
                Font font = new Font( "Arial", 18 );                    // 3
                SolidBrush brush = new SolidBrush( Color.Black );       // 4
                float x = 100.0F;
                float y = 20.0F;
                g.DrawString( s, font, brush, x, y );
                font.Dispose();                                         // 5
                brush.Dispose();                                        // 6
        }
    }
  }
}
```

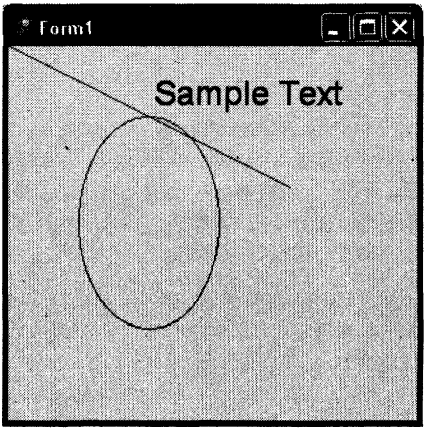


Рис. 14.13. Форма с графикой

Графические объекты потребляют системные ресурсы, поэтому рекомендуется вызывать для них метод освобождения ресурсов `Dispose`. Для упрощения работы с такими объектами в C# есть *оператор* `using`[1] со следующим синтаксисом:

```
using ( выделение_ресурса ) оператор
```

Под *ресурсом* здесь понимается объект, который реализует интерфейс `System.IDisposable`, включающий метод `Dispose`. Код, использующий ресурс, вызовом этого метода сигнализирует о том, что ресурс больше не требуется. Если метод `Dispose` не был вызван, освобождение ресурса выполняется в процессе сборки мусора.

Оператор `using` неявным образом вызывает метод `Dispose` в случае успешного создания и использования объекта. Этот способ применен в операторах 1 и 2.

[1] Ключевое слово `using` используется в C# в двух не связанных между собой случаях: как директива и как оператор. Директива `using` была рассмотрена в главе 12.

В операторах 3 и 4 объекты создаются обычным образом, поэтому для них требуется явный вызов Dispose, что и происходит в операторах 5 и 6.

Как видно даже из этого простого листинга, для вывода графики требуется кропотливое изучение множества свойств и методов множества стандартных классов, описание которых, во-первых, очень объемное, во-вторых, невыносимо скучное, а в-третьих, не входит в задачу учебника по основам программирования.

Рекомендации по программированию

Процесс создания Windows-приложения состоит из двух основных этапов, которые могут чередоваться между собой: это визуальное проектирование приложения и определение его поведения.

При задании внешнего облика приложения следует обратить внимание на стандарты интерфейса Windows-приложений: компания Microsoft, в свое время заимствовавшая идею стандартного графического интерфейса у компании Apple, довела эту идею до совершенства, детально регламентировав вид окон, расположение, цветовую гамму и пропорции компонентов.

Основная сложность для начинающих заключается в разработке алгоритма: по каким событиям будут выполняться действия, реализующие функциональность программы, какие действия должны выполняться при щелчке на кнопках, вводе текста, выборе пунктов меню и т. д.

Интерфейс программы должен быть интуитивно понятным и по возможности простым. Часто повторяющиеся действия не должны требовать от пользователя выполнения сложных последовательностей операций. Команды меню и компоненты, которые не имеет смысла использовать в данный момент, рекомендуется делать неактивными. Вопросы, задаваемые пользователю программы, должны быть ненавязчивыми («Нет, а все-таки Вы действительно хотите удалить этот файл?») и немногословными, но при этом не допускать двояких толкований.

Эта глава получилась самой длинной из-за большого количества информации справочного характера. Несмотря на это приведенных сведений совершенно недостаточно для создания реальных Windows-приложений. К сожалению, мощь библиотеки .NET имеет оборотную сторону: для освоения необходимой информации требуется много времени и упорства, однако это единственный путь для тех, кто хочет заниматься программированием профессионально.

Конечно, пытаться запомнить все методы и свойства классов нет смысла, достаточно изучить состав используемых пространств имен, представлять себе возможности их элементов и знать, как быстро найти требуемую информацию.

Для дальнейшего изучения возможностей библиотеки можно рекомендовать документацию и дополнительную литературу [17], [18], [20], [31]. И последний совет: не следует считать себя программистом только на том основании, что вы умеете размещать компоненты на форме!

Глава 15
Дополнительные средства C#

В этой главе описаны дополнительные средства языка C# и среды Visual Studio: указатели, регулярные выражения и документация в формате XML. В конце главы дается краткое введение в основные области профессионального применения C#: ASP.NET (веб-формы и веб-службы) и ADO.NET (базы данных).

Указатели, без которых не мыслят свою жизнь программисты, использующие C и C++, в языке C# рекомендуется применять только в случае необходимости, поскольку они сводят на нет многие преимущества этого языка. Документирование кода в формате XML и регулярные выражения применяются шире, но относятся скорее к дополнительным возможностям языка, поэтому не были рассмотрены ранее.

Напротив, веб-формы, веб-службы и работа с базами данных являются одними из основных областей применения C#, но не рассматриваются в этой книге из-за того, что подобные темы обычно не входят в базовый курс программирования, поскольку для их полноценного освоения требуется иметь базовые знания в области сетей, баз данных, протоколов передачи данных и т. п.

Небезопасный код

Одним из основных достоинств языка C# является его схема работы с памятью: автоматическое выделение памяти под объекты и автоматическая уборка мусора. При этом невозможно обратиться по несуществующему адресу памяти или выйти за границы массива, что делает программы более надежными и безопасными и исключает возможность появления целого класса ошибок, доставляющих массу неудобств при написании программ на других языках.

Однако в некоторых случаях возникает необходимость работать с адресами памяти непосредственно, например, при взаимодействии с операционной системой, написании драйверов или программ, время выполнения которых критично. Такую возможность предоставляет так называемый *небезопасный* (unsafe) *код*.

Небезопасным называется код, выполнение которого среда CLR не контролирует. Он работает напрямую с адресами областей памяти посредством *указателей* и должен быть явным образом помечен с помощью ключевого слова unsafe, которое определяет так называемый *небезопасный контекст* выполнения.

Ключевое слово unsafe может использоваться либо как *спецификатор*, либо как *оператор*. В первом случае его указывают наряду с другими спецификаторами при описании класса, делегата, структуры, метода, поля и т. д. — везде, где допустимы другие спецификаторы. Это определяет небезопасный контекст для описываемого элемента, например:

```
public unsafe struct Node
{
   public int   Value;
   public Node* Left;
   public Node* Right;
}
```

Вся структура Node помечается как небезопасная, что делает возможным использование в ней указателей Left и Right. Можно применить и другой вариант описания, в котором небезопасными объявляются только соответствующие поля структуры:

```
public struct Node
{
   public int Value;
   public unsafe Node* Left;
   public unsafe Node* Right;
}
```

Оператор unsafe имеет следующий синтаксис:

```
unsafe блок
```

Все операторы, входящие в блок, выполняются в небезопасном контексте.

ПРИМЕЧАНИЕ

Компиляция кода, содержащего небезопасные фрагменты, должна производиться с ключом /unsafe. Этот режим можно установить путем настройки среды Visual Studio (Project ▸ Properties ▸ Configuration Properties ▸ Build ▸ Allow Unsafe Code).

Синтаксис указателей

Указатели предназначены для хранения адресов областей памяти. Синтаксис объявления указателя:

```
тип* переменная;
```

Здесь *тип* — это тип величины, на которую указывает *переменная*, то есть величины, хранящейся по записанному в переменной адресу. Тип не может быть классом, но может быть структурой, перечислением, указателем, а также одним

из стандартных типов: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool и void. Последнее означает, что указатель ссылается на переменную неизвестного типа.

Указатель на тип void применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Указателю на тип void можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом.

Примеры объявления указателей:

```
int*   a;          // указатель на int
Node*  pNode;      // указатель на описанную ранее структуру Node
void*  p;          // указатель на неопределенный тип
int*[] m;          // одномерный массив указателей на int
int**  d;          // указатель на указатель на int
```

В одном операторе можно описать несколько указателей одного и того же типа, например:

```
int* a, b, c;      // три указателя на int
```

Указатели являются отдельной категорией типов данных. Они не наследуются от типа object, и преобразование между типом object и типами указателей невозможно. В частности, для них не выполняется упаковка и распаковка. Однако допускаются преобразования между разными типами указателей, а также указателями и целыми.

ПРИМЕЧАНИЕ

Именно потому что указатели могут содержать адрес любой области памяти и, следовательно, изменить ее, операции с ними и называются небезопасными.

Величины типа указателя могут являться локальными переменными, полями, параметрами и возвращаемым значением функции. Эти величины подчиняются общим правилам определения области действия и времени жизни.

Преобразования указателей

Для указателей поддерживаются неявные преобразования из любого типа указателя к типу void*. Любому указателю можно присвоить константу null. Кроме того, допускаются явные преобразования:

- между указателями любого типа;
- между указателями любого типа и целыми типами (со знаком и без знака).

Корректность преобразований лежит на совести программиста. Преобразования никак не влияют на величины, на которые ссылаются указатели, но при попытке получения значения по указателю несоответствующего типа поведение программы не определено[1].

Инициализация указателей

Ниже перечислены способы присваивания значений указателям:

1. Присваивание указателю адреса существующего объекта:
   - с помощью операции получения адреса:

```
int  a = 5;      // целая переменная
int* p = &a;     // в указатель записывается адрес a
```

ПРИМЕЧАНИЕ

Программист должен сам следить за тем, чтобы переменные, на которые ссылается указатель, были правильно инициализированы.

   - с помощью значения другого инициализированного указателя:

```
int* r = p;
```

   - с помощью имени массива, которое трактуется как адрес:

```
int[] b = new int[] {10, 20, 30, 50};                    // массив
fixed ( int* t = b ) { ... };  // присваивание адреса начала массива
fixed ( int* t = &b[0] ) { ... };                        // то же самое
```

ПРИМЕЧАНИЕ

Оператор `fixed` рассматривается позже.

2. Присваивание указателю адреса области памяти в явном виде:

```
char* v = (char *)0x12F69E;
```

Здесь `0x12F69E` — шестнадцатеричная константа, `(char *)` — операция приведения типа: константа преобразуется к типу указателя на `char`.

ПРИМЕЧАНИЕ

Использовать этот способ можно только в том случае, если адрес вам точно известен, в противном случае может возникнуть исключение.

3. Присваивание нулевого значения:

```
int* xx = null;
```

[1] Как правило, если в документации встречается оборот «неопределенное поведение» (undefined behavior), ничего хорошего это не сулит.

4. Выделение области памяти в стеке и присваивание ее адреса указателю:

```
int* s = stackalloc int [10];
```

Здесь операция `stackalloc` выполняет выделение памяти под 10 величин типа `int` (массив из 10 элементов) и записывает адрес начала этой области памяти в переменную `s`, которая может трактоваться как имя массива.

ПРИМЕЧАНИЕ

Специальных операций для выделения области динамической памяти (хипа) в C# не предусмотрено. В случае необходимости можно использовать, например, системную функцию `HeapAlloc` (пример см. в спецификации языка).

Операции с указателями

Все операции с указателями выполняются в небезопасном контексте. Они перечислены в табл. 15.1.

Таблица 15.1. Операции с указателями

Операция	Описание
*	Разадресация — получение значения, которое находится по адресу, хранящемуся в указателе
->	Доступ к элементу структуры через указатель
[]	Доступ к элементу массива через указатель
&	Получение адреса переменной
++, --	Увеличение и уменьшение значения указателя на один адресуемый элемент
+, -	Сложение с целой величиной и вычитание указателей
==, !=, <>, <=, >=	Сравнение адресов, хранящихся в указателях. Выполняется как сравнение беззнаковых целых величин
stackalloc	Выделение памяти в стеке под переменную, на которую ссылается указатель

Рассмотрим примеры применения операций. Если в указатель занесен адрес объекта, получить доступ к этому объекту можно с помощью операций разадресации и доступа к элементу.

Операция разадресации, или *разыменования*, предназначена для доступа к величине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины, например:

```
int  a = 5;                          // целая переменная
int* p = &a;                         // инициализация указателя адресом a
Console.WriteLine( *p );             // операция разадресации, результат: 5
Console.WriteLine( ++(*p) );         // результат: 6
int[] b = new int[] {10, 20, 30, 50}; // массив
```

```
fixed ( int* t = b )       // инициализация указателя адресом начала массива
{
    int* z = t;            // инициализация указателя значением другого указателя
    for (int i = 0; i < b.Length; ++i )
    {
        t[i] += 5;     // доступ к элементу массива (увеличение на 5)
        *z += 5;       // доступ с помощью разадресации (увеличение еще на 5)
         ++z;          // инкремент указателя
    }
    Console.WriteLine( &t[5] - t );         // операция вычитания указателей
}
```

Оператор `fixed` фиксирует объект, адрес которого заносится в указатель, для того чтобы его не перемещал сборщик мусора и, таким образом, указатель остался корректным. Фиксация происходит на время выполнения блока, который записан после круглых скобок.

В приведенном примере доступ к элементам массива выполняется двумя способами: путем индексации указателя `t` и путем разадресации указателя `z`, значение которого инкрементируется при каждом проходе цикла для перехода к следующему элементу массива.

Конструкцию `*переменная` можно использовать в левой части оператора присваивания, так как она определяет адрес области памяти. Для простоты эту конструкцию можно считать именем переменной, на которую ссылается указатель. С ней допустимы все действия, определенные для величин соответствующего типа.

Арифметические операции с указателями (сложение с целым, вычитание, инкремент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, элементы которых размещены в памяти последовательно, например, с массивами.

Инкремент перемещает указатель к следующему элементу массива, *декремент* — к предыдущему. Фактически значение указателя изменяется на величину `sizeof (тип)`, где `sizeof` — операция получения размера величины указанного типа (в байтах). Эта операция применяется только в небезопасном контексте, с ее помощью можно получать размеры не только стандартных, но и пользовательских типов данных. Для структуры результат может быть больше суммы длин составляющих ее полей из-за выравнивания элементов.

Если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа, например:

```
short* p; ...
p++;                          // значение p увеличивается на 2
long* q; ...
q++;                          // значение q увеличивается на 4
```

Разность двух указателей — это разность их значений, деленная на размер типа в байтах. Так, результат выполнения последней операции вывода в приведенном примере равен 5. Суммирование двух указателей не допускается.

При записи выражений с указателями следует обращать внимание на приоритеты операций. В качестве примера рассмотрим последовательность действий, заданную в операторе

```
*p++ = 10;
```

Поскольку инкремент постфиксный, он выполняется после выполнения операции присваивания. Таким образом, сначала по адресу, записанному в указателе p, будет записано значение 10, а затем указатель увеличится на количество байтов, соответствующее его типу. То же самое можно записать подробнее:

```
*p = 10; p++;
```

Выражение (*p)++, напротив, инкрементирует значение, на которое ссылается указатель.

В следующем примере каждый байт беззнакового целого числа x выводится на консоль с помощью указателя t:

```
uint x = 0xAB10234F;
byte* t = (byte*)&x;
for ( int i = 0; i < 4; ++i )
    Console.Write("{0:X} ", *t++ );        // результат: 4F 23 10 AB
```

Как видите, первоначально указатель t был установлен на младший байт переменной x. Листинг 15.1 иллюстрирует доступ к полю класса и элементу структуры:

Листинг 15.1. Доступ к полю класса и элементу структуры с помощью указателей

```
using System;
namespace ConsoleApplication1
{
    class A
    {    public int value = 20;    }

    struct B
    {    public int a;    }

     class Program
     {
       unsafe static void Main()
        {
           A n = new A();
           fixed ( int* pn = &n.value ) ++(*pn);
           Console.WriteLine( "n = " + n.value );          // результат: 21

           B b;
           B* pb = &b;
```

продолжение

Листинг 15.1 (*продолжение*)

```
            pb->a = 100;
            Console.WriteLine( b.a );                    // результат: 100
        }
    }
}
```

Операция stackalloc позволяет выделить память в стеке под заданное количество величин заданного типа:

```
stackalloc тип [ количество ]
```

Количество задается целочисленным выражением. Если памяти недостаточно, генерируется исключение System.StackOverflowException. Выделенная память ничем не инициализируется и автоматически освобождается при завершении блока, содержащего эту операцию. Пример выделения памяти под пять элементов типа int и их заполнения числами от 0 до 4:

```
int* p = stackalloc int [5];
for ( int i = 0; i < 5; ++i )
{
    p[i] = i;
    Console.Write( p[i] + " " );          // результат: 0 1 2 3 4
}
```

В листинге 15.2 приведен пример работы с указателями, взятый из спецификации C#. Метод IntToString преобразует переданное ему целое значение в строку символов, заполняя ее посимвольно путем доступа через указатель.

Листинг 15.2. Пример работы с указателями: перевод числа в строку

```
using System;
class Test
{
    static string IntToString ( int value )
    {
        int n = value >= 0 ? value : -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)( n % 10 + '0' );
                n /= 10;
            } while ( n != 0 );
            if ( value < 0 ) *--p = '-';
            return new string( p, 0, (int)( buffer + 16 - p ) );
        }
    }
    static void Main() {
        Console.WriteLine( IntToString( 12345 ) );
        Console.WriteLine( IntToString( -999 ) );
    }
}
```

Регулярные выражения

Регулярные выражения предназначены для обработки текстовой информации и обеспечивают:

- эффективный поиск в тексте по заданному шаблону;
- редактирование, замену и удаление подстрок;
- формирование итоговых отчетов по результатам работы с текстом.

С помощью регулярных выражений удобно обрабатывать, например, файлы в формате HTML, файлы журналов или длинные текстовые файлы. Для поддержки регулярных выражений в библиотеку .NET включены классы, объединенные в пространство имен `System.Text.RegularExpressions`.

Метасимволы

Регулярное выражение — это шаблон (образец), по которому выполняется поиск соответствующего ему фрагмента текста. Язык описания регулярных выражений состоит из символов двух видов: обычных и *метасимволов*. Обычный символ представляет в выражении сам себя, а метасимвол — некоторый класс символов, например любую цифру или букву.

Например, регулярное выражение для поиска в тексте фрагмента «Вася» записывается с помощью четырех обычных символов `Вася`, а выражение для поиска двух цифр, идущих подряд, состоит из двух метасимволов `\d\d`.

С помощью комбинаций метасимволов можно описывать сложные шаблоны для поиска. Например, можно описать шаблон для IP-адреса, адреса электронной почты, различных форматов даты, заголовков определенного вида и т. д.

ПРИМЕЧАНИЕ

Синтаксис регулярных выражений .NET в основном позаимствован из языка Perl 5. Неподготовленного человека вид сложного регулярного выражения может привести в замешательство, но при вдумчивом изучении он обязательно почувствует его красоту и очарование. Пожалуй, регулярные выражения более всего напоминают заклинания, по которым волшебным образом преобразуется текст. Ошибка всего в одном символе делает заклинание бессильным, зато, верно составленное, оно творит чудеса!

В табл. 15.2 описаны наиболее употребительные метасимволы, представляющие собой классы символов.

Метасимволы, перечисленные в табл. 15.3, уточняют позицию в строке, в которой следует искать совпадение с регулярным выражением, например, только в начале или в конце строки. Эти метасимволы являются мнимыми, то есть в тексте им не соответствует никакой реальный символ.

Таблица 15.2. Классы символов

Класс символов	Описание	Пример
.	Любой символ, кроме \n	Выражение c.t соответствует фрагментам cat, cut, c1t, c{t и т. д.
[]	Любой одиночный символ из последовательности, записанной внутри скобок. Допускается использование диапазонов символов	Выражение c[aul]t соответствует фрагментам cat, cut и c1t, а выражение c[a-z]t — фрагментам cat, cbt, cct, cdt, ..., czt
[^]	Любой одиночный символ, не входящий в последовательность, записанную внутри скобок. Допускается использование диапазонов символов	Выражение c[^aul]t соответствует фрагментам cbt, c2t, cXt и т. д., а выражение c[^a-zA-Z]t — фрагментам cит, c1t, cЧt, c3t и т. д.
\w	Любой алфавитно-цифровой символ, то есть символ из множества прописных и строчных букв и десятичных цифр	Выражение c\wt соответствует фрагментам cat, cut, c1t, c0t и т. д., но не соответствует фрагментам c{t, c;t и т. д.
\W	Любой *не* алфавитно-цифровой символ, то есть символ, не входящий в множество прописных и строчных букв и десятичных цифр	Выражение c\Wt соответствует фрагментам c{t, c;t, c t и т. д., но не соответствует фрагментам cat, cut, c1t, c0t и т. д.
\s	Любой пробельный символ, например символ пробела, табуляции (\t, \v), перевода строки (\n, \r), новой страницы (\f)	Выражение \s\w\w\w\s соответствует любому слову из трех букв, окруженному пробельными символами
\S	Любой *не* пробельный символ, то есть символ, не входящий в множество пробельных	Выражение \s\S\S\s соответствует любым двум непробельным символам, окруженным пробельными
\d	Любая десятичная цифра	Выражение c\dt соответствует фрагментам c1t, c2t, ..., c9t
\D	Любой символ, не являющийся десятичной цифрой	Выражение c\Dt не соответствует фрагментам c1t, c2t, ..., c9t

Таблица 15.3. Уточняющие метасимволы

Метасимвол	Описание
^	Фрагмент, совпадающий с регулярным выражением, следует искать только в начале строки
$	Фрагмент, совпадающий с регулярным выражением, следует искать только в конце строки
\A	Фрагмент, совпадающий с регулярным выражением, следует искать только в начале многострочной строки

Метасимвол	Описание
\Z	Фрагмент, совпадающий с регулярным выражением, следует искать только в конце многострочной строки
\b	Фрагмент, совпадающий с регулярным выражением, начинается или заканчивается на границе слова (то есть между символами, соответствующими метасимволам \w и \W)
\B	Фрагмент, совпадающий с регулярным выражением, не должен встречаться на границе слова

Например, выражение ^cat соответствует символам cat, встречающимся в начале строки, выражение cat$ — символам cat, встречающимся в конце строки (то есть за ними идет символ перевода строки), а выражение ^$ представляет пустую строку, то есть начало строки, за которым сразу же следует ее конец.

В регулярных выражениях часто используют повторители. *Повторители* — это метасимволы, которые располагаются непосредственно после обычного символа или класса символов и задают количество его повторений в выражении. Например, если требуется записать выражение для поиска в тексте пяти идущих подряд цифр, вместо метасимволов \d\d\d\d\d можно записать \d{5}. Такому выражению будут соответствовать фрагменты 11111, 12345, 53332 и т. д.

Наиболее употребительные повторители перечислены в табл. 15.4.

Таблица 15.4. Повторители

Метасимвол	Описание	Пример
*	Ноль или более повторений предыдущего элемента	Выражение ca*t соответствует фрагментам ct, cat, caat, caaaaaaaaaat и т. д.
+	Одно или более повторений предыдущего элемента	Выражение ca+t соответствует фрагментам cat, caat, caaaaaaaaaat и т. д.
?	Ни одного или одно повторение предыдущего элемента	Выражение ca?t соответствует фрагментам ct и cat
{n}	Ровно n повторений предыдущего элемента	Выражение ca{3}t соответствует фрагменту caaat, а выражение (cat){2} — фрагменту catcat[1]
{n,}	По крайней мере n повторений предыдущего элемента	Выражение ca{3,}t соответствует фрагментам caaat, caaaat, caaaaaaaaaaat и т. д.
{n,m}	От n до m повторений предыдущего элемента	Выражение ca{2,4}t соответствует фрагментам caat, caaat и caaaat

Помимо рассмотренных элементов регулярных выражений можно использовать конструкцию *выбора* из нескольких элементов. Варианты выбора перечисляются

[1] Круглые скобки служат для группировки символов.

через вертикальную черту. Например, если требуется определить, присутствует ли в тексте хотя бы один элемент из списка «cat», «dog» и «horse», можно использовать выражение

```
cat|dog|horse
```

При поиске используется так называемый «ленивый» алгоритм, по которому поиск прекращается при нахождении самого короткого из возможных фрагментов, совпадающих с регулярным выражением.

Примеры простых регулярных выражений:

- целое число (возможно, со знаком):

```
[-+]?\d+
```

- вещественное число (может иметь знак и дробную часть, отделенную точкой):

```
[-+]?\d+\.?\d*
```

- российский номер автомобиля (упрощенно):

```
[A-Z]\d{3}[A-Z]{2}\d\dRUS
```

ВНИМАНИЕ

Если требуется описать в выражении обычный символ, совпадающий с каким-либо метасимволом, его предваряют обратной косой чертой. Так, для поиска в тексте символа точки следует записать \., а для поиска косой черты — \\.

Например, для поиска в тексте имени файла cat.doc следует использовать регулярное выражение `cat\.doc`. Символ «точка» *экранируется* обратной косой чертой для того, чтобы он воспринимался не как метасимвол «любой символ» (в том числе и точка!), а непосредственно[1].

Для группирования элементов выражения используются круглые скобки. *Группирование* применяется во многих случаях, например, если требуется задать повторитель не для отдельного символа, а для последовательности (это использовано в предыдущей таблице). Кроме того, группирование служит для запоминания в некоторой переменной фрагмента текста, совпавшего с выражением, заключенным в скобки. Имя переменной задается в угловых скобках или апострофах:

```
(?<имя_переменной>фрагмент_выражения)
```

Фрагмент текста, совпавший при поиске с фрагментом регулярного выражения, заносится в переменную с заданным именем. Пусть, например, требуется выделить из текста номера телефонов, записанных в виде `nnn-nn-nn`. Регулярное выражение для поиска номера можно записать так:

```
(?<num>\d\d\d-\d\d-\d\d)
```

[1] Обратите внимание на то, что метасимволы регулярных выражений не совпадают с метасимволами, которые используются в шаблонах имен файлов, таких как *.doc.

При анализе текста в переменную с именем num будут последовательно записываться найденные номера телефонов.

Рассмотрим еще один вариант применения группирования — для формирования *обратных ссылок*. Все конструкции, заключенные в круглые скобки, автоматически нумеруются, начиная с 1. Эти номера, предваренные обратной косой чертой, можно использовать для ссылок на соответствующую конструкцию. Например, выражение (\w)\1 используется для поиска сдвоенных символов в словах (wa*ll*, ma*ss*, co*o*perate)[1].

Круглые скобки могут быть вложенными, при этом номер конструкции определяется порядком открывающей скобки в выражении. Примеры:

```
(Вася)\s+(должен)\s+(?<sum>\d+)\sруб\.\s+Ну что же ты, \1
```

В этом выражении три подвыражения, заключенных в скобки. Ссылка на первое из них выполняется в конце выражения. С этим выражением совпадут, например, фрагменты

```
Вася должен 5 руб. Ну что же ты, Вася
Вася    должен    53459 руб.    Ну что же ты, Вася
```

Выражение, задающее IP-адрес:

```
((\d{1,3}\.){3}\d{1,3})
```

Адрес состоит из четырех групп цифр, разделенных точками. Каждая группа может включать от одной до трех цифр. Примеры IP-адресов: 212.46.197.69, 212.194.5.106, 209.122.173.160. Первая группа, заключенная в скобки, задает весь адрес. Ей присваивается номер 1. В нее вложены вторые скобки, определяющие границы для повторителя {3}.

Переменную, имя которой задается внутри выражения в угловых скобках, также можно использовать для обратных ссылок в последующей части выражения. Например, поиск двойных символов в словах можно выполнить с помощью выражения (?<s>\w)\k<s>, где s — имя переменной, в которой запоминается символ, \k — элемент синтаксиса.

В регулярное выражение можно помещать *комментарии*. Поскольку выражения обычно проще писать, чем читать, это — очень полезная возможность. Комментарий либо помещается внутрь конструкции (?#), либо располагается, начиная от символа # до конца строки[2].

Классы библиотеки .NET для работы с регулярными выражениями

Классы библиотеки .NET для работы с регулярными выражениями объединены в пространство имен System.Text.RegularExpressions.

---

[1] Если написать просто \w\w, будут найдены все пары алфавитно-цифровых символов.

[2] Для распознавания этого вида комментария должен быть включен режим x RegexOptions.IgnorePatternWhitespace.

Начнем с класса Regex, представляющего собственно регулярное выражение. Класс является неизменяемым, то есть после создания экземпляра его корректировка не допускается. Для описания регулярного выражения в классе определено несколько перегруженных *конструкторов*:

- Regex() — создает пустое выражение;
- Regex(String) — создает заданное выражение;
- Regex(String, RegexOptions) — создает заданное выражение и задает параметры для его обработки с помощью элементов перечисления RegexOptions (например, различать или не различать прописные и строчные буквы).

Пример конструктора, задающего выражение для поиска в тексте повторяющихся слов, расположенных подряд и разделенных произвольным количеством пробелов, независимо от регистра:

```
Regex rx = new Regex( @"\b(?<word>\w+)\s+(\k<word>)\b",
                      RegexOptions.IgnoreCase );
```

Поиск фрагментов строки, соответствующих заданному выражению, выполняется с помощью методов IsMatch, Match и Matches.

Метод IsMatch возвращает true, если фрагмент, соответствующий выражению, в заданной строке найден, и false в противном случае. В листинге 15.3 приведен пример поиска повторяющихся слов в двух тестовых строках. В регулярное выражение, приведенное ранее, добавлен фрагмент, позволяющий распознавать знаки препинания.

Листинг 15.3. Поиск в строке дублированных слов (методом IsMatch)

```
using System;
using System.Text.RegularExpressions;

public class Test
{   public static void Main()
    {
        Regex r = new Regex( @"\b(?<word>\w+)[.,:;!? ]\s*(\k<word>)\b",
                             RegexOptions.IgnoreCase );

        string tst1 = "Oh, oh! Give me more!";
        if ( r.IsMatch( tst1 ) ) Console.WriteLine( " tst1 yes" );
        else                     Console.WriteLine( " tst1 no"  );

        string tst2 = "Oh give me, give me more!";
        if ( r.IsMatch( tst2 ) ) Console.WriteLine( " tst2 yes" );
        else                     Console.WriteLine( " tst2 no"  );
    }
}
```

Результат работы программы:

```
tst1 yes
tst2 no
```

Повторяющиеся слова в строке tst2 располагаются не подряд, поэтому она не соответствует регулярному выражению. Для поиска повторяющихся слов, расположенных в произвольных позициях строки, в регулярном выражении нужно всего-навсего заменить пробел (\s) «любым символом» (.):

```
Regex r = new Regex( @"\b(?<word>\w+)[.,:;!? ].*(\k<word>)\b",
                     RegexOptions.IgnoreCase );
```

Метод Match класса Regex, в отличие от метода IsMatch, не просто определяет, произошло ли совпадение, а возвращает объект класса Match — очередной фрагмент, совпавший с образцом. Рассмотрим листинг 15.4, в котором используется этот метод.

Листинг 15.4. Выделение из строки слов и чисел (методом Match)

```
using System;
using System.Text.RegularExpressions;
public class Test
{
    public static void Main()
    {
        string text    = "Салат - $4, борщ - $3, одеколон - $10.";
        string pattern = @"(\w+) - \$(\d+)[.,]";
        Regex  r       = new Regex( pattern );
        Match  m       = r.Match( text );
        int    total   = 0;
        while ( m.Success )
        {
            Console.WriteLine( m );
            total += int.Parse( m.Groups[2].ToString() );
            m = m.NextMatch();
        }
        Console.WriteLine( "Итого: $" + total );
    }
}
```

Результат работы программы:

```
Салат - $4,
борщ - $3,
одеколон - $10.
Итого: $17
```

При первом обращении к методу Match возвращается первый фрагмент строки, совпавший с регулярным выражением pattern. В классе Match определено свойство Groups, возвращающее коллекцию фрагментов, совпавших с подвыражениями в круглых скобках. Нулевой элемент коллекции содержит весь фрагмент, первый элемент — фрагмент, совпавший с подвыражением в первых скобках, второй элемент — фрагмент, совпавший с подвыражением во вторых скобках, и т. д. Если при определении выражения задать фрагментам имена, как это было

сделано в предыдущем листинге, можно будет обратиться к ним по этим именам, например:

```
string pattern = @"(?'name'\w+) - \$(?'price'\d+)[.,]";
...
total += int.Parse( m.Groups["price"].ToString() );
```

ПРИМЕЧАНИЕ

Метод NextMatch класса Match продолжает поиск в строке с того места, на котором закончился предыдущий поиск.

Метод Matches класса Regex возвращает объект класса MatchCollection — коллекцию всех фрагментов заданной строки, совпавших с образцом.

Рассмотрим теперь пример применения метода Split класса Regex. Этот метод разбивает заданную строку на фрагменты в соответствии с разделителями, заданными с помощью регулярного выражения, и возвращает эти фрагменты в массиве строк. В листинге 15.5 строка из листинга 15.4 разбивается на отдельные слова.

Листинг 15.5. Разбиение строки на слова (методом Split)

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;
public class Test
{   public static void Main()
    {
        string text        = "Салат -  $4, борщ -$3, одеколон - $10.";
        string pattern     = "[- ,.]+";
        Regex  r           = new Regex( pattern );
        List<string> words = new List<string>( r.Split( text ) );
        foreach ( string word in words ) Console.WriteLine( word );
    }
}
```

Результат работы программы:

```
Салат
$4
борщ
$3
одеколон
$10
```

Метод Replace класса Regex позволяет выполнять замену фрагментов текста. Определено несколько перегруженных версий этого метода. Вот как выглядит пример простейшего применения метода в его статическом варианте, заменяющего все вхождения символа $ символами у.е.:

```
string text  = "Салат -  $4, борщ -$3, одеколон - $10.";
string text1 = Regex.Replace( text, @"\$", "у.е." );
```

Другие версии метода позволяют задавать любые действия по замене с помощью делегата MatchEvaluator, который вызывается для каждого вхождения фрагмента, совпавшего с заданным регулярным выражением.

ПРИМЕЧАНИЕ

Помимо классов Regex и Match в пространстве имен System.Text.RegularExpressions определены вспомогательные классы, например, класс Capture — фрагмент, совпавший с подвыражением в круглых скобках; класс CaptureCollection — коллекция фрагментов, совпавших со всеми подвыражениями в текущей группе; класс Group содержит коллекцию Capture для текущего совпадения с регулярным выражением и т. д.

В качестве более реального примера применения регулярных выражений рассмотрим программу анализа файла журнала веб-сервера. Это текстовый файл, каждая строка которого содержит информацию об одном соединении с сервером. Четыре строки файла приведены ниже:

```
ppp-48.pool-113.spbnit.ru - - [31/May/2002:02:08:32 +0400] "GET / HTTP/1.1" 200
2434 "http://www.price.ru/bin/price/firminfo_f?fid=10922&where=01&base=2"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
81.24.130.7 - - [31/May/2002:08:13:17 +0400] "GET /swf/menu.swf HTTP/1.1" 200
4682 "-" "Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)"
81.24.130.7 - - [31/May/2002:08:13:17 +0400] "GET /swf/header.swf HTTP/1.1" 200
21244 "-" "Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)"
gate.solvo.ru - - [31/May/2002:10:43:03 +0400] "GET / HTTP/1.0" 200 2422
"http://www.price.ru/bin/price/firminfo_f?fid=10922&where=01&base=1"
"Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)"
```

Подобные файлы могут иметь весьма значительный объем, поэтому составление итогового отчета в удобном для восприятия формате имеет важное значение. Если рассматривать каждую строку файла как совокупность полей, разделенных пробелами, то поле номер 0 содержит адрес, с которого выполнялось соединение с сервером, поле номер 5 — операцию (GET при загрузке информации), поле 8 — признак успешности выполнения операции (200 — успешно) и, наконец, поле 9 — количество переданных байтов.

Приведенная в листинге 15.6 программа формирует в формате HTML итоговый отчет, содержащий таблицу адресов, с которых выполнялось обращение к серверу, и суммарное количество переданных байтов для каждого адреса.

Листинг 15.6. Анализ журнала веб-сервера

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Text.RegularExpressions;

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

продолжение

Листинг 15.6 (*продолжение*)

```
        StreamReader f   = new StreamReader( "access_log" );
        StreamWriter w   = new StreamWriter( "report.htm" );
        Regex        get = new Regex( "GET" );
        Regex        r   = new Regex( " " );
        string s, entry;
        int value;
        string[] items   = new string[40];
        Dictionary<string, int> table = new Dictionary<string, int>();

        while ( (s = f.ReadLine() ) != null )
        {
            items = r.Split( s );
            if ( get.IsMatch( items[5] ) && items[8] == "200" )
            {
                entry = items[0];
                value = int.Parse( items[9] );
                if ( table.ContainsKey( entry ) ) table[entry] += value;
                else                              table[entry]  = value;
            }
        }
        f.Close();
        w.Write( "<html><head><title> Report </title></head><body>" +
                 "<table border =1 <tr><td> Computer <td> Bytes </tr>" );
        foreach ( string item in table.Keys )
            w.Write( "<tr><td>{0}<td>{1}</tr>", item, table[item] );
        w.Write( "</table></body></html>" );
        w.Close();
    }
}
```

Фрагмент результата работы программы показан на рис. 15.1.

Computer	Bytes
ppp-48.pool-113.spbnit.ru	107039
test223.sovam.com	2422
210.82.124.83	20052
ipblock209-209-002.octetgroup.net	162781
81.24.130.7	74756
gate.solvo.ru	113692
212.113.108.164	261199

Рис. 15.1. Фрагмент журнала веб-сервера

Файл access_log считывается построчно, каждая строка разбивается на поля, которые заносятся в массив items. Затем, если загрузка прошла успешно, о чем свидетельствуют значения GET и 200 полей 5 и 8, количество переданных байтов

(поле 9) преобразуется в целое и заносится в хеш-таблицу по ключу, которым служит адрес, хранящийся в поле 0.

Для формирования HTML-файла report.htm используются соответствующие теги. Файл можно просмотреть, например, с помощью Internet Explorer. Как видите, программа получилась весьма компактной за счет использования стандартных классов библиотеки .NET[1].

Документирование в формате XML

XML (eXtensible Markup Language) — это язык разметки текста. Разметкой является все, что не относится к содержанию, или, как модно говорить, контенту: структура документа, формат, вид и т. д. Разметка осуществляется с помощью *тегов* — управляющих элементов, заключенных в угловые скобки. Теги в XML всегда парные: открывающий тег записывается перед размечаемым фрагментом, а закрывающий — после него. Закрывающий тег выглядит так же, как открывающий, но предваряется косой чертой, например:

```
<summary> Класс для работы с регулярными выражениями </summary>
```

В тегах могут присутствовать *атрибуты* — элементы вида `имя = значение`, уточняющие и дополняющие описание элемента.

Язык XML широко распространен в Интернете благодаря его универсальности и переносимости. Корпоративные приложения используют XML как основной формат для обмена данными. Строго говоря, XML является не языком, а системой правил для описания языков.

ПРИМЕЧАНИЕ

Многие составляющие технологии .NET неразрывно связаны с XML, поэтому в пространстве имен `System.Xml` библиотеки .NET описано множество классов, поддерживающих XML. Объем и задача учебника не позволяют описать эти классы и технологии.

Любой программный продукт требуется документировать. Соответствие версий документации и программы — серьезная проблема, которая решается в .NET встраиванием документации прямо в код программы с помощью комментариев и XML-тегов.

Комментарии, используемые для построения файлов документации, начинаются с символов /// и размещаются перед соответствующим элементом программы. Внутри комментариев записываются теги, относящиеся к комментируемому элементу — классу, методу, параметру метода и т. п. Теги перечислены в табл. 15.5.

[1] Справедливости ради надо отметить, что аналогичная программа на языке Perl примерно в два раза короче.

Таблица 15.5. Теги документирования

Тег	Описание
<c>	Форматирование как фрагмента кода
<code>	Многострочный код (используется в секции <example>)
<example>	Пример использования класса или метода
<exception>	Исключение, генерируемое классом
<include file='файл' path='путь[@name="ид"]'' />	Ссылка на комментарии в другом файле, в котором находятся описания элементов исходного кода
<list>	Перечисление в виде списка
<param>	Описание параметра метода
<paramref>	Ссылка на параметр
<permission>	Права доступа к элементу
<remarks>	Подробное описание элемента (класса, метода и т. п.)
<returns>	Возвращаемое значение метода
<see cref="элемент">	Ссылка на элемент класса
<seealso cref="элемент">	Ссылка на документацию вида «см. также»
<summary>	Краткое описание элемента (класса, метода и т. п.)
<value>	Описание значения свойства

Примеры тегов:

```
///<summary> Функция вычисления синуса </summary>
///<param name="i"> Аргумент функции </param>
///<seealso cref="System.Double"> double </seealso>
///<returns> Возвращает величину синуса </returns>
///<remarks> Для вычислений используется разложение
///в ряд Тейлора </remarks>
public double Sin( double i ) { … }
```

Для построения файлов документации в формате XML требуется использовать режим компиляции `/doc:имя_файла.xml`. Этот режим можно установить и в среде Visual Studio в окне свойств проекта (Project ▸ Properties) на вкладке Build (задать имя файла XML documentation file). Компилятор проверяет правильность записи тегов и их соответствие элементам программы.

ПРИМЕЧАНИЕ

В Visual Studio 2003 для построения файлов документации в формате HTML используется команда меню Tools ▸ Build Comment Web Pages.

Темы, не рассмотренные в книге

Эта книга заканчивается там, где начинается самое интересное — профессиональное применение C#: создание веб-форм и веб-служб, распределенных приложений, работа с базами данных с помощью ADO.NET и т. д. Этим вопросам посвящены

тысячи страниц книг и документации, а здесь приводится краткое введение для того, чтобы облегчить вам выбор материала для дальнейшего изучения.

ADO.NET

Работа с данными является одной из главных задач при создании как сетевых, так и автономных приложений. Библиотека .NET содержит богатый набор средств под общим названием ADO.NET (ActiveX Data Objects), поддерживающих взаимодействие с локальными и удаленными хранилищами данных.

Объектная модель ADO.NET состоит из классов двух видов: *компоненты сущностей* (content components) и *компоненты управляемых поставщиков* (managed-provider components). Основным классом первого вида является класс `DataSet`, представляющий собой набор связанных таблиц — локальную копию базы данных или ее части. Кроме того, определены вспомогательные классы `DataTable`, `DataRow`, `DataColumn` и `DataRelation`. В классах этого вида располагаются пересылаемые данные. Класс `DataSet` может содержать несколько объектов `DataTable` и `DataRelation`. В классе `DataSet` описан набор методов, интегрирующих его с XML, что делает возможным межплатформенное взаимодействие.

Компоненты управляемых поставщиков обеспечивают интерфейс для доступа к данным (извлечения и обновления). Для непосредственной работы с данными используются объекты `Connection`, `Command` и `DataReader`. Класс `DataAdapter` играет роль канала передачи данных между хранилищем и компонентами сущностей.

Данные могут представлять собой выборку из базы данных, XML-файл или, например, таблицу Excel.

Классы ADO.NET предназначены для решения следующих задач:

- установления соединения с хранилищем данных;
- создания и заполнения данными объекта `DataSet`;
- отключения от хранилища данных;
- возврата изменений, внесенных в `DataSet`, обратно в хранилище данных.

Классы ADO.NET определены в пространствах имен `System.Data`, `System.Data.Common`, `System.Data.OleDb`, `System.Data.SqlClient` и `System.Data.SqlTypes`.

Среда Visual Studio .NET располагает средствами, упрощающими программирование баз данных. В среду включена копия ядра MSDE. С помощью окна Server Explorer (View ▸ Server Explorer) можно подключиться к SQL Server в локальной или удаленной системе. После подключения можно выполнять различные операции с базами данных, таблицами и хранимыми процедурами.

ASP.NET

Под термином ASP.NET (Active Server Pages for .NET) объединяются все средства поддержки веб-серверов в .NET, включая веб-страницы и веб-службы.

Сервер — это аппаратный или программный компонент вычислительной системы, выполняющий специализированные функции по запросу клиента, предоставляя

ему доступ к определенным ресурсам. Сервер, реализованный в виде программы или программного модуля, обычно решает строго определенную задачу и обменивается информацией с клиентом по определенному протоколу. Примеры программных серверов: FTP-сервер, веб-сервер (Apache, IIS), сервер баз данных, почтовый сервер.

Веб-сервер — это сервер, предоставляющий доступ к сайтам World Wide Web. Когда пользователь дает браузеру команду открыть документ на некотором сайте, браузер подключается к соответствующему серверу и запрашивает у него содержимое документа. Обычно веб-сервер работает по протоколам HTTP и/или HTTPS. На сегодня наиболее распространенными веб-серверами являются:

- Apache (свободно распространяемый веб-сервер с открытым исходным кодом; наиболее часто используется в Unix-подобных операционных системах);
- IIS (Internet Information Services) от компании Microsoft.

ПРИМЕЧАНИЕ

IIS поставляется Microsoft как часть операционной системы, но по умолчанию в Windows 2000 Professional не устанавливается. Для установки IIS воспользуйтесь командой меню Пуск ▸ Настройка ▸ Панель управления ▸ Установка и удаление программ ▸ Установка компонентов Windows. После этого потребуется зарегистрировать его с помощью утилиты aspnet_regiis, следуя инструкциям справочной службы.

Веб-приложение — это набор взаимосвязанных файлов, расположенных на IIS-сервере в своем *виртуальном каталоге*, которому соответствует физический каталог на диске. Файлы веб-страниц имеют расширение aspx. Для создания веб-приложения следует выбрать шаблон ASP.NET Web Application. Обратите внимание на то, что в поле Location записан URL-адрес компьютера, а не путь к каталогу на диске. Вид среды после создания проекта практически такой же, как и при создании Windows-приложения, однако для разработки интерфейса веб-страницы используются элементы категории Web Form Controls, основанные на HTML-коде, а не категории Windows Forms.

Интерактивная веб-страница создается так же, как обычное Windows-приложение: перетаскиванием элементов управления с панели инструментов на форму, настройкой их характеристик в окне свойств и заданием реакции на события. Среда автоматически создает файл для генерации HTML-кода с расширением aspx (его можно просмотреть на вкладке HTML окна редактора кода) и связанный с ним файл на языке C# с расширением aspx.cs. В этом файле расположено описание класса, являющегося потомком `System.Web.UI.Page`. Страница (aspx-файл) содержит ссылку на этот класс. Когда клиент запрашивает страницу, среда выполнения ASP.NET создает экземпляр класса.

Возможность применения стандартных элементов управления из категории Web Form Controls является одним из важнейших достоинств ASP.NET, поскольку при этом значительно упрощается создание пользовательского интерфейса на веб-страницах. С элементами управления можно работать и как с обычными классами C#, и через aspx-файл. В каждом элементе определены набор событий, которые будут обрабатываться на сервере, и средства проверки ввода данных пользователем.

ПРИМЕЧАНИЕ

В ASP.NET есть два элемента, `DataGrid` и `DataList`, которые предназначены для отображения данных, полученных из источника (обычно это объект ADO `DataSet`). С помощью этих объектов можно создать приложение для решения одной из часто встречающихся задач — найти в каком-либо источнике данные по запросу пользователя и вернуть их в виде таблицы.

С помощью XML можно создавать программные компоненты, которые взаимодействуют друг с другом независимо от языка и платформы. Веб-службы обеспечивают доступ к программным компонентам через стандартные протоколы Интернета, такие как HTTP и SMTP.

Веб-служба .NET — это модуль кода .NET, который обычно устанавливается на IIS-сервере. Веб-служба строится из тех же типов, что и любая сборка .NET: классов, интерфейсов, перечислений и структур, которые для клиента представляют собой «черный ящик», отвечающий на запросы. Службы предназначены для обработки удаленных вызовов, поэтому у них обычно отсутствует графический интерфейс пользователя.

Одно из значений термина «служба» в обычной жизни — это, например, справочная служба или служба быта, когда мы по запросу получаем какую-либо услугу от поставщика услуг. В программном обеспечении службой называется блок кода, способный выполнить какие-либо действия по запросу пользователя (считать данные из источника, выполнить вычисления) и ждать следующего запроса. Веб-службы могут использоваться любым приложением, умеющим разбирать XML-поток, переданный по протоколу HTTP.

Веб-служба, как и обычное приложение ASP.NET, традиционно располагается в виртуальном каталоге на IIS-сервере. Файл веб-службы имеет расширение `asmx`. В нем так же, как и в aspx-файле, содержится ссылка на кодовый файл на языке C# с расширением `asm.cs`, в котором и находится собственно код веб-службы. Класс, обеспечивающий работу веб-службы, является потомком класса `System.Web.Services.WebService`. Для создания веб-службы используется шаблон проекта **ASP.NET Web Service**.

Заключение

Чтобы использовать язык C# на профессиональном уровне, необходимо не только хорошо представлять себе его конструкции, но и изучить бесчисленные классы библиотеки .NET. В этом вам поможет электронная документация и книги, например, [13], [20], [21], [23], [27], [31]–[33]. Для понимания возможностей классов необходимо представлять себе основы построения сетей и баз данных, протоколы Интернета, HTML, XML и многое другое.

Желаю вам получить удовольствие от погружения в это бескрайнее море информации и от программирования на прекрасном языке! Ведь, говоря высоким слогом, познание и созидание — одни из самых сильных и глубоких радостей жизни.

Лабораторные работы

Лабораторная работа 1. Линейные программы

Теоретический материал: главы 1–3.

Напишите программу для расчета по двум формулам. Предварительно подготовьте тестовые примеры с помощью калькулятора (результаты вычисления по обеим формулам должны совпадать). Класс `Math`, содержащий математические функции C#, описан на с. 64. Кроме того, для поиска нужной функции можно воспользоваться алфавитным указателем. Методы, отсутствующие в классе, выразите через имеющиеся.

1. $z_1 = \cos\alpha + \sin\alpha + \cos 3\alpha + \sin 3\alpha;$ $z_2 = \frac{1}{4} - \frac{1}{4}\sin\left(\frac{5}{2}\pi - 8\alpha\right).$
2. $z_1 = \cos\alpha + \sin\alpha + \cos 3\alpha + \sin 3\alpha;$ $z_2 = 2\sqrt{2}\cos\alpha \cdot \sin\left(\frac{\pi}{4} + 2\alpha\right).$
3. $z_1 = \frac{\sin 2\alpha + \sin 5\alpha - \sin 3\alpha}{\cos\alpha + 1 - 2\sin^2 2\alpha};$ $z_2 = 2\sin\alpha.$
4. $z_1 = \frac{\sin 2\alpha + \sin 5\alpha - \sin 3\alpha}{\cos\alpha - \cos 3\alpha + \cos 5\alpha};$ $z_2 = \operatorname{tg} 3\alpha.$
5. $z_1 = 1 - \frac{1}{4}\sin^2 2\alpha + \cos 2\alpha;$ $z_2 = \cos^2\alpha + \cos^4\alpha.$
6. $z_1 = \cos\alpha + \cos 2\alpha + \cos 6\alpha + \cos 7\alpha;$ $z_2 = 4\cos\frac{\alpha}{2} \cdot \cos\frac{5}{2}\alpha \cdot \cos 4\alpha.$
7. $z_1 = \cos^2\left(\frac{3}{8}\pi - \frac{\alpha}{4}\right) - \cos^2\left(\frac{11}{8}\pi + \frac{\alpha}{4}\right);$ $z_2 = \frac{\sqrt{2}}{2}\sin\frac{\alpha}{2}.$
8. $z_1 = \cos^4 x + \sin^2 y + \frac{1}{4}\sin^2 2x - 1;$ $z_2 = \sin(y + x) \cdot \sin(y - x).$

9. $z_1 = (\cos\alpha - \cos\beta)^2 - (\sin\alpha - \sin\beta)^2$; $z_2 = -4\sin^2\dfrac{\alpha-\beta}{2}\cdot\cos(\alpha+\beta)$.

10. $z_1 = \left(\sin\left(\dfrac{\pi}{2}+3\alpha\right)\right)\Big/(1-\sin(3\alpha-\pi))$; $z_2 = \operatorname{ctg}\left(\dfrac{5}{4}\pi+\dfrac{3}{2}\alpha\right)$.

11. $z_1 = \dfrac{1-2\sin^2\alpha}{1+\sin 2\alpha}$; $z_2 = \dfrac{1-\operatorname{tg}\alpha}{1+\operatorname{tg}\alpha}$.

12. $z_1 = \dfrac{\sin 4\alpha}{1+\cos 4\alpha}\dfrac{\cos 2\alpha}{1+\cos 2\alpha}$; $z_2 = \operatorname{ctg}\left(\dfrac{3}{2}\pi-\alpha\right)$.

13. $z_1 = \dfrac{\sin\alpha+\cos(2\beta-\alpha)}{\cos\alpha-\sin(2\beta-\alpha)}$; $z_2 = \dfrac{1+\sin 2\beta}{\cos 2\beta}$.

14. $z_1 = \dfrac{\cos\alpha+\sin\alpha}{\cos\alpha-\sin\alpha}$; $z_2 = \operatorname{tg} 2\alpha + \sec 2\alpha$.

15. $z_1 = \dfrac{\sqrt{2b+2\sqrt{b^2-4}}}{\sqrt{b^2-4}+b+2}$; $z_2 = \dfrac{1}{\sqrt{b+2}}$.

16. $z_1 = \dfrac{x^2+2x-3+(x+1)\sqrt{x^2-9}}{x^2-2x-3+(x-1)\sqrt{x^2-9}}$; $z_2 = \sqrt{\dfrac{x+3}{x-3}}$.

17. $z_1 = \dfrac{\sqrt{(3m+2)^2-24m}}{3\sqrt{m}-\dfrac{2}{\sqrt{m}}}$; $z_2 = -\sqrt{m}$.

18. $z_1 = \left(\dfrac{a+2}{\sqrt{2a}}-\dfrac{a}{\sqrt{2a}+2}+\dfrac{2}{a-\sqrt{2a}}\right)\dfrac{\sqrt{a}-\sqrt{2}}{a+2}$; $z_2 = \dfrac{1}{\sqrt{a}+\sqrt{2}}$.

19. $z_1 = \left(\dfrac{1+a+a^2}{2a+a^2}+2-\dfrac{1-a+a^2}{2a-a^2}\right)^{-1}(5-2a^2)$; $z_2 = \dfrac{4-a^2}{2}$.

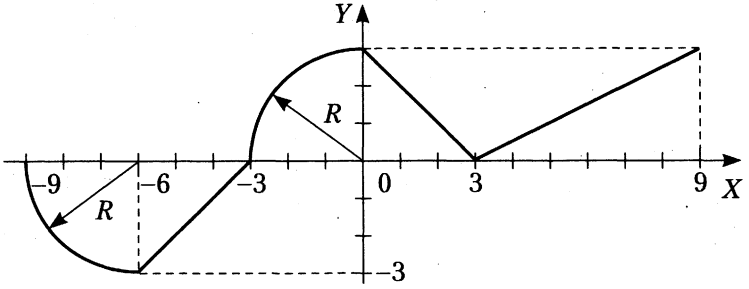
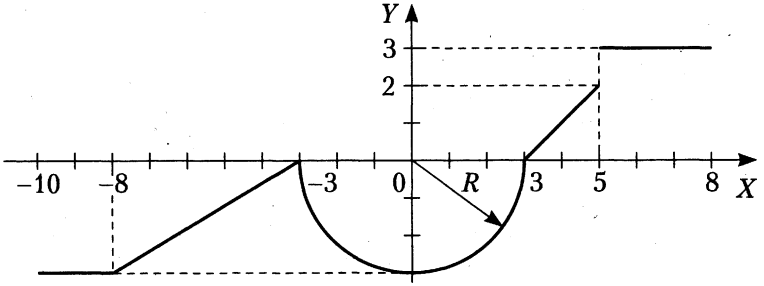
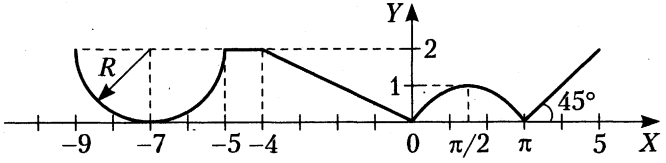
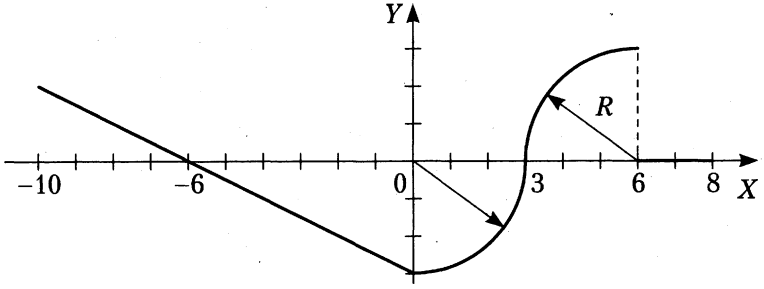
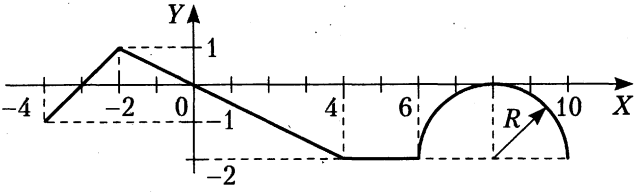
20. $z_1 = \dfrac{(m-1)\sqrt{m}-(n-1)\sqrt{n}}{\sqrt{m^3n}+nm+m^2-m}$; $z_2 = \dfrac{\sqrt{m}-\sqrt{n}}{m}$.

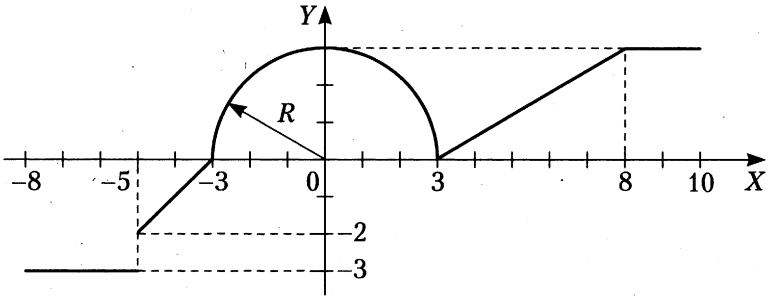
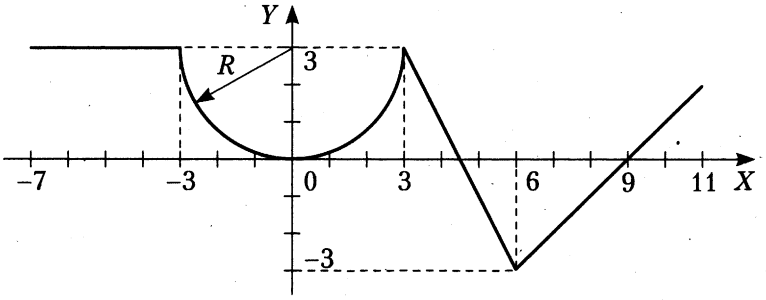
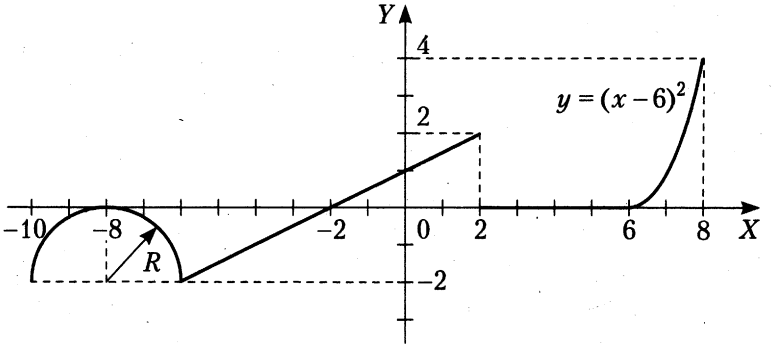
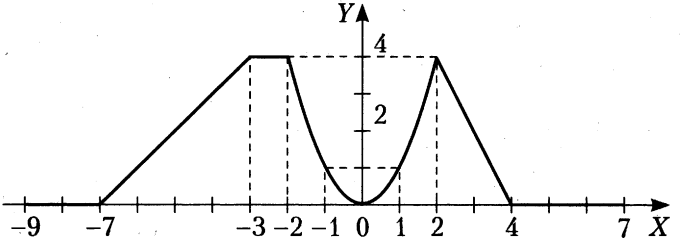
Лабораторная работа 2. Разветвляющиеся вычислительные процессы

Теоретический материал: глава 4, раздел «Операторы ветвления».

Задание 1. Вычисление значения функции

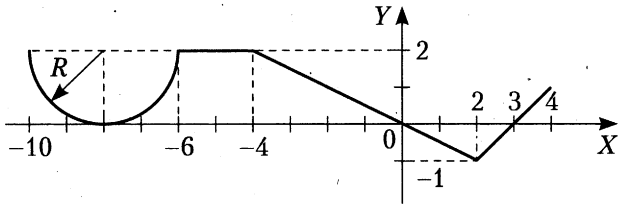
Написать программу, которая по введенному значению аргумента вычисляет значение функции, заданной в виде графика. Параметр R вводится с клавиатуры.

№	Графики
1	
2	
3	
4	
5	

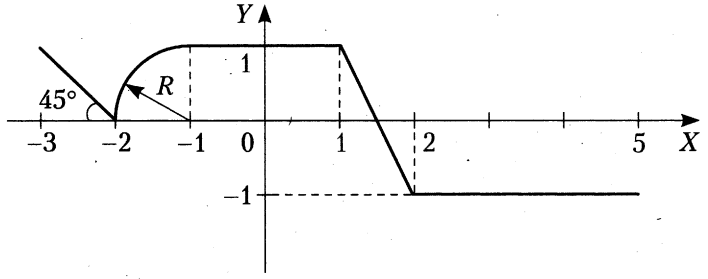
№	Графики
6	
7	
8	
9	

№	Графики

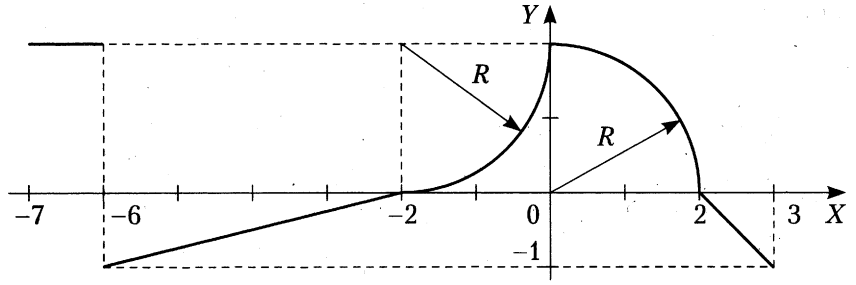
10



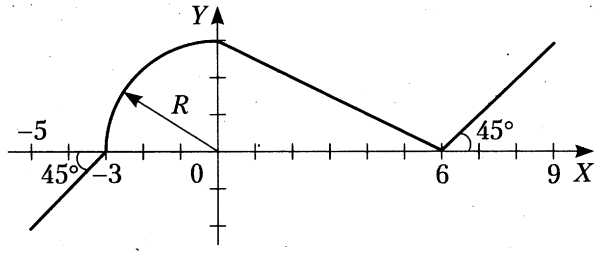
11



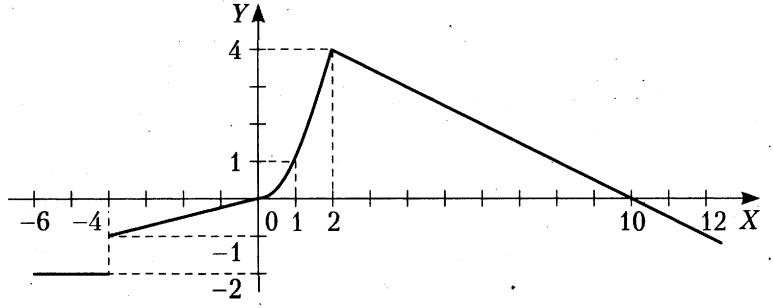
12

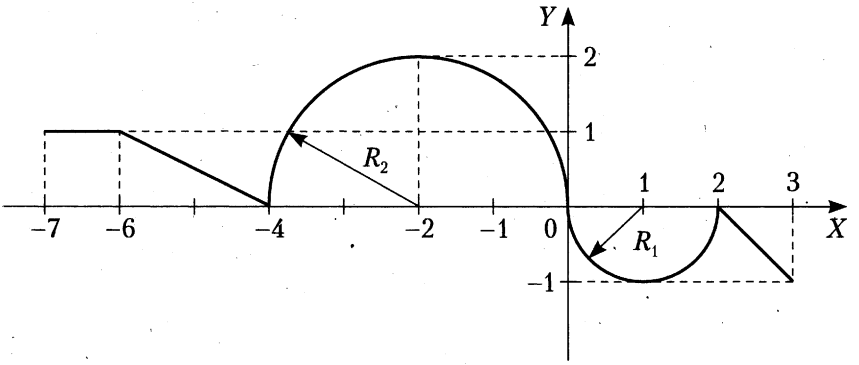
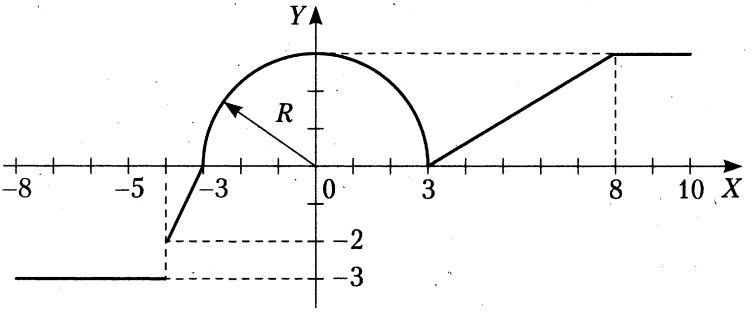
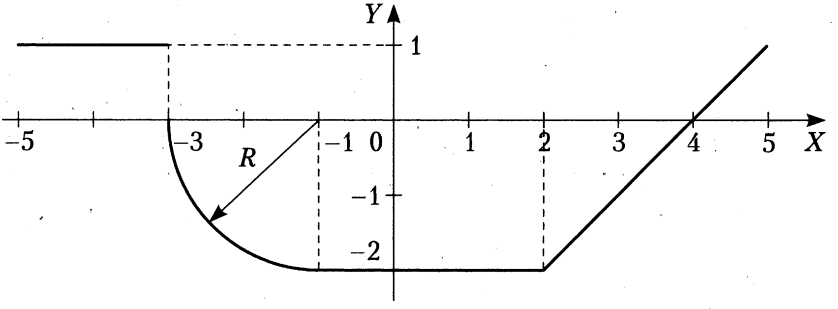
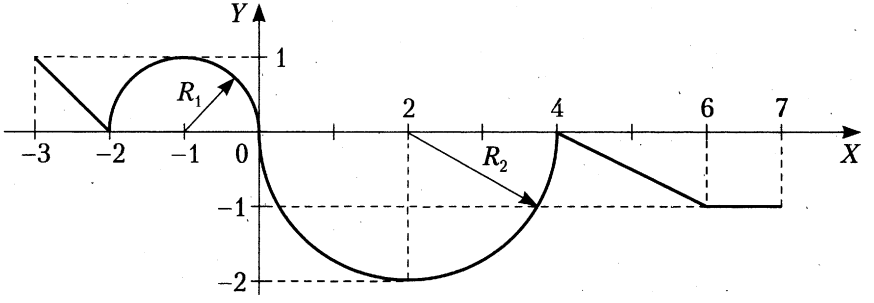


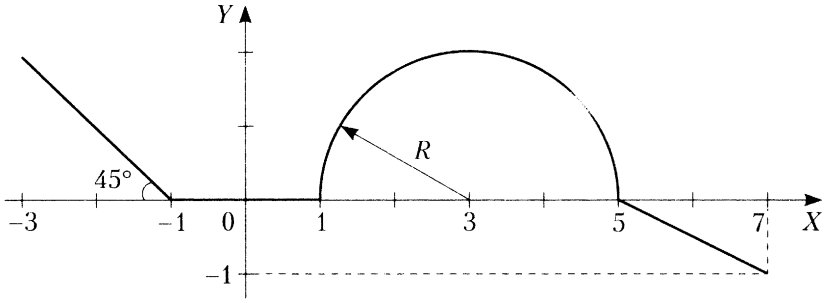
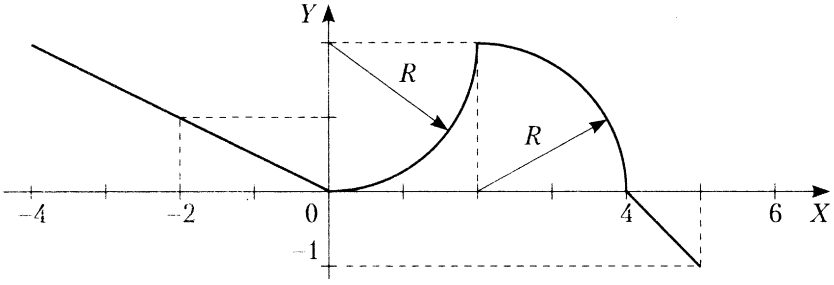
13



14

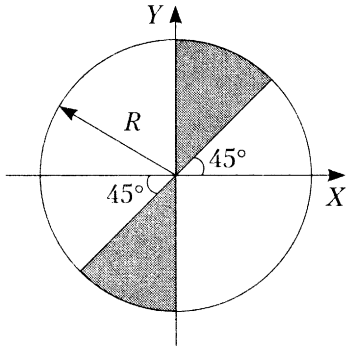
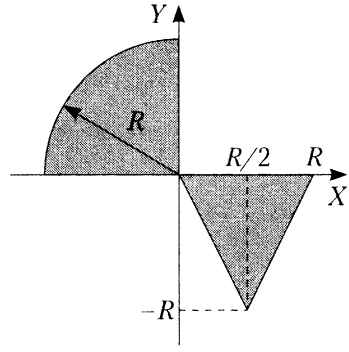


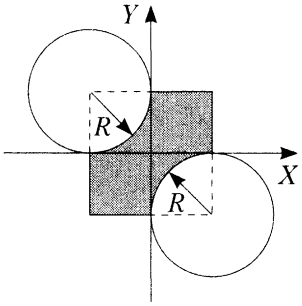
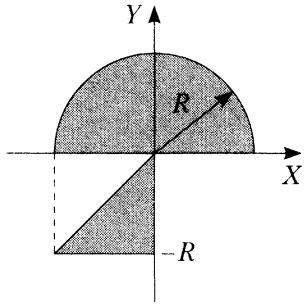
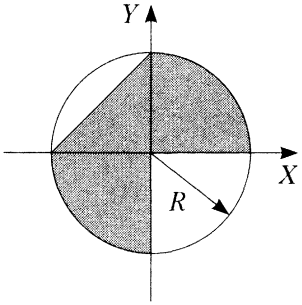
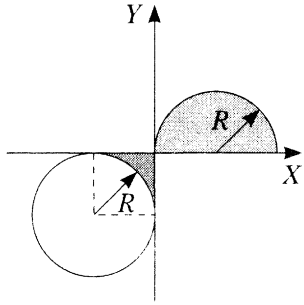
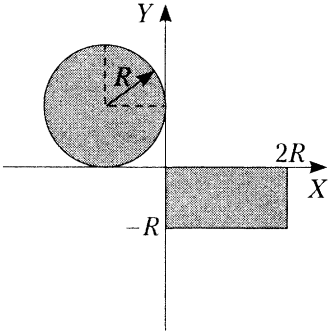
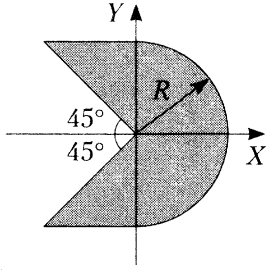
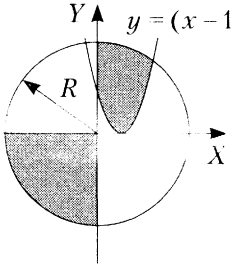
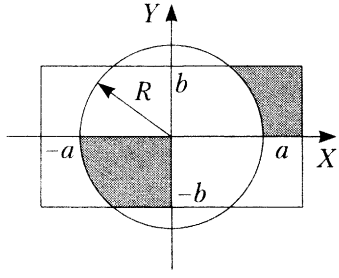
№	Графики
15	
16	
17	
18	

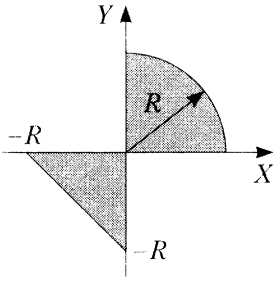
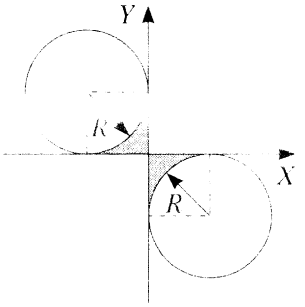
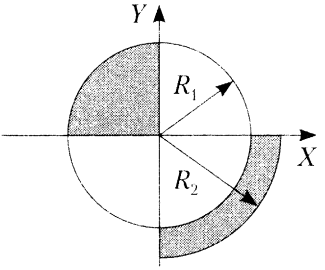
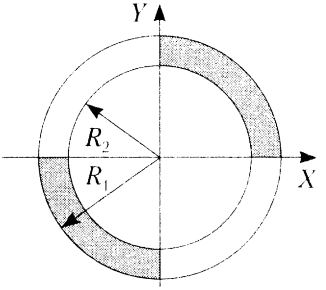
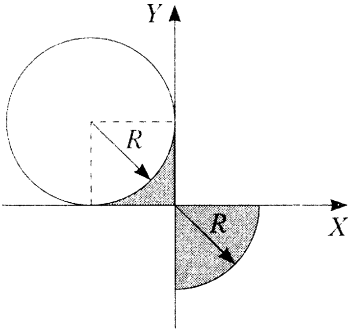
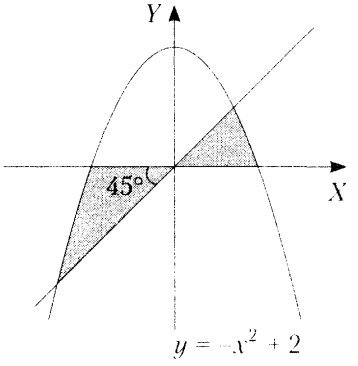
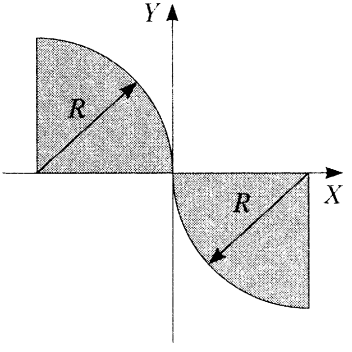
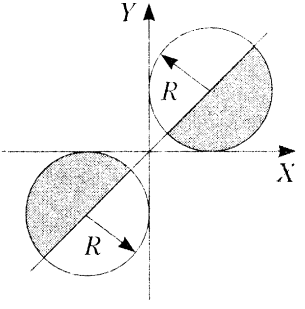
№	Графики
19	
20	

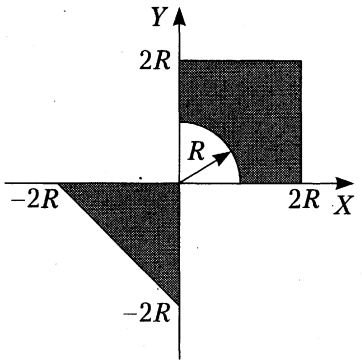
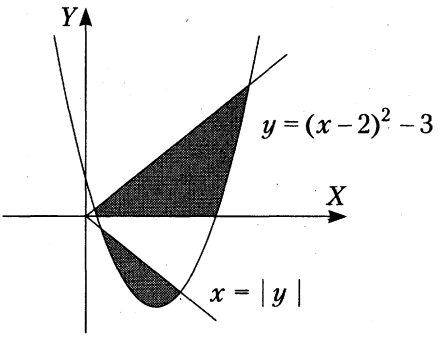
Задание 2. Попадание точки в заштрихованную область

Написать программу, которая определяет, попадает ли точка с заданными координатами в область, закрашенную на рисунке серым цветом. Результат работы программы вывести в виде текстового сообщения.

№	Область	№	Область
1		2	

№	Область	№	Область
3		4	
5		6	
7		8	
9		10	

№	Область	№	Область
11		12	
13		14	
15		16	
17		18	

№	Область	№	Область
19		20	

Лабораторная работа 3. Организация циклов

Теоретический материал: глава 4, разделы «Операторы цикла», «Базовые конструкции структурного программирования».

Задание 1. Таблица значений функции

Вычислить и вывести на экран в виде таблицы значения функции, заданной графически (см. задание 1 лабораторной работы 2), на интервале от $x_{нач}$ до $x_{кон}$ с шагом dx. Интервал и шаг задать таким образом, чтобы проверить все ветви программы. Таблицу снабдить заголовком и шапкой.

Задание 2. Серия выстрелов по мишени

Для десяти выстрелов, координаты которых задаются с клавиатуры, вывести текстовые сообщения о попадании в мишень из задания 2 лабораторной работы 2.

Задание 3. Ряды Тейлора

Вычислить и вывести на экран в виде таблицы значения функции, заданной с помощью ряда Тейлора, на интервале от $x_{нач}$ до $x_{кон}$ с шагом dx с точностью ε. Таблицу снабдить заголовком и шапкой. Каждая строка таблицы должна содержать значение аргумента, значение функции и количество просуммированных членов ряда.

1. $\ln\dfrac{x+1}{x-1} = 2\sum\limits_{n=0}^{\infty}\dfrac{1}{(2n+1)x^{2n+1}} = 2\left(\dfrac{1}{x} + \dfrac{1}{3x^3} + \dfrac{1}{5x^5} + \ldots\right),\ |x| > 1.$

2. $e^{-x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \ldots, |x| < \infty.$

3. $e^{x} = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots, |x| < \infty.$

4. $\ln(x+1) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{n+1}}{n+1} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} - \ldots, -1 < x < 1.$

5. $\ln\frac{1+x}{1-x} = 2\sum_{n=0}^{\infty} \frac{x^{2n+1}}{2n+1} = 2\left(x + \frac{x^3}{3} + \frac{x^5}{5} + \ldots\right), |x| < 1.$

6. $\ln(1-x) = -\sum_{n=1}^{\infty} \frac{x^n}{n} = -\left(x + \frac{x^2}{2} + \frac{x^3}{3} + \ldots\right), -1 <= x < 1.$

7. $\operatorname{arcctg} x = \frac{\pi}{2} + \sum_{n=0}^{\infty} \frac{(-1)^{n+1} x^{2n+1}}{2n+1} = \frac{\pi}{2} - x + \frac{x^3}{3} - \frac{x^5}{5} - \ldots, |x| < 1.$

8. $\operatorname{arctg} x = \frac{\pi}{2} + \sum_{n=0}^{\infty} \frac{(-1)^{n+1}}{(2n+1)x^{2n+1}} = \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \ldots, x > 1.$

9. $\operatorname{arctg} x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots, |x| < 1.$

10. $\operatorname{arth} x = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{2n+1} = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \ldots, |x| < 1.$

11. $\operatorname{arth} x = \sum_{n=0}^{\infty} \frac{1}{(2n+1)x^{2n+1}} = \frac{1}{x} + \frac{1}{3x^3} + \frac{1}{5x^5} + \ldots, |x| > 1.$

12. $\operatorname{arctg} x = -\frac{\pi}{2} + \sum_{n=0}^{\infty} \frac{(-1)^{n+1}}{(2n+1)x^{2n+1}} = -\frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \ldots, x < -1.$

13. $e^{-x^2} = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{n!} = 1 - x^2 + \frac{x^4}{2!} - \frac{x^6}{3!} + \frac{x^8}{4!} - \ldots, |x| < \infty.$

14. $\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, |x| < \infty.$

15. $\frac{\sin x}{x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n+1)!} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \ldots, |x| < \infty.$

16. $\ln x = 2\sum_{n=0}^{\infty} \frac{(x-1)^{2n+1}}{(2n+1)(x+1)^{2n+1}} = 2\left(\frac{x-1}{x+1} + \frac{(x-1)^3}{3(x+1)^3} + \frac{(x-1)^5}{5(x+1)^5} + \ldots\right), x > 0.$

17. $\ln x = \sum_{n=0}^{\infty} \frac{(x-1)^{n+1}}{(n+1)x^{n+1}} = \frac{x-1}{x} + \frac{(x-1)^2}{2x^2} + \frac{(x-1)^3}{3x^3} + \ldots, x > \frac{1}{2}.$

18. $\ln x = \sum_{n=0}^{\infty} \frac{(-1)^n (x-1)^{n+1}}{(n+1)} = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \ldots, 0 < x < 2.$

19. $\arcsin x = x + \sum_{n=1}^{\infty} \frac{1 \cdot 3 \cdots (2n-1) \cdot x^{2n+1}}{2 \cdot 4 \cdots 2n \cdot (2n+1)} =$

$= x + \frac{x^3}{2 \cdot 3} + \frac{1 \cdot 3 \cdot x^5}{2 \cdot 4 \cdot 5} + \frac{1 \cdot 3 \cdot 5 \cdot x^7}{2 \cdot 4 \cdot 6 \cdot 7} + \frac{1 \cdot 3 \cdot 5 \cdot 7 \cdot x^9}{2 \cdot 4 \cdot 6 \cdot 8 \cdot 9} \ldots, \quad |x| < 1.$

20. $\arccos x = \frac{\pi}{2} - \left(x + \sum_{n=1}^{\infty} \frac{1 \cdot 3 \cdots (2n-1) \cdot x^{2n+1}}{2 \cdot 4 \cdots 2n \cdot (2n+1)} \right) =$

$= \frac{\pi}{2} - \left(x + \frac{x^3}{2 \cdot 3} + \frac{1 \cdot 3 \cdot x^5}{2 \cdot 4 \cdot 5} + \frac{1 \cdot 3 \cdot 5 \cdot x^7}{2 \cdot 4 \cdot 6 \cdot 7} + \frac{1 \cdot 3 \cdot 5 \cdot 7 \cdot x^9}{2 \cdot 4 \cdot 6 \cdot 8 \cdot 9} \ldots \right), \quad |x| < 1.$

Лабораторная работа 4. Простейшие классы

Теоретический материал: глава 4, раздел «Обработка исключительных ситуаций», глава 5.

Каждый разрабатываемый класс должен, как правило, содержать следующие элементы: скрытые поля, конструкторы с параметрами и без параметров, методы, свойства. Методы и свойства должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. При возникновении ошибок должны выбрасываться исключения.

В программе должна выполняться проверка всех разработанных элементов класса.

Вариант 1

Описать класс, реализующий десятичный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и свойство, позволяющее получить его текущее состояние. При выходе за границы диапазона выбрасываются исключения.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 2

Описать класс, реализующий шестнадцатеричный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и свойство, позволяющее получить его текущее состояние. При выходе за границы диапазона выбрасываются исключения.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 3

Описать класс, представляющий треугольник. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и вращения на заданный угол. Описать свойства для получения состояния объекта. При невозможности построения треугольника выбрасывается исключение.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 4

Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса и проверки допустимости вводимых значений. В случае недопустимых значений полей выбрасываются исключения.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 5

Составить описание класса для представления комплексных чисел. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 6

Составить описание класса для вектора, заданного координатами его концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 7

Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменение размеров, построение наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 8

Составить описание класса для представления даты. Предусмотреть возможности установки даты и изменения ее отдельных полей (год, месяц, день) с проверкой допустимости вводимых значений. В случае недопустимых значений полей выбрасываются исключения. Создать методы изменения даты на заданное количество дней, месяцев и лет.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 9

Составить описание класса для представления времени. Предусмотреть возможности установки времени и изменения его отдельных полей (час, минута, секунда) с проверкой допустимости вводимых значений. В случае недопустимых значений полей выбрасываются исключения. Создать методы изменения времени на заданное количество часов, минут и секунд.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 10

Составить описание класса многочлена вида $ax^2 + bx + c$. Предусмотреть методы, реализующие:

- вычисление значения многочлена для заданного аргумента;
- операцию сложения, вычитания и умножения многочленов с получением нового объекта-многочлена;
- вывод на экран описания многочлена.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 11

Описать класс, представляющий треугольник. Предусмотреть методы для создания объектов, вычисления площади, периметра и точки пересечения медиан. Описать свойства для получения состояния объекта. При невозможности построения треугольника выбрасывается исключение.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 12

Описать класс, представляющий круг. Предусмотреть методы для создания объектов, вычисления площади круга, длины окружности и проверки попадания заданной точки внутрь круга. Описать свойства для получения состояния объекта.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 13

Описать класс для работы со строкой, позволяющей хранить только двоичное число и выполнять с ним арифметические операции. Предусмотреть инициализацию с проверкой допустимости значений. В случае недопустимых значений выбрасываются исключения.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 14

Описать класс дробей — рациональных чисел, являющихся отношением двух целых чисел. Предусмотреть методы сложения, вычитания, умножения и деления дробей.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 15

Описать класс «файл», содержащий сведения об имени, дате создания и длине файла. Предусмотреть инициализацию с проверкой допустимости значений полей. В случае недопустимых значений полей выбрасываются исключения. Описать метод добавления информации в конец файла и свойства для получения состояния файла.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 16

Описать класс «комната», содержащий сведения о метраже, высоте потолков и количестве окон. Предусмотреть инициализацию с проверкой допустимости значений полей. В случае недопустимых значений полей выбрасываются исключения. Описать методы вычисления площади и объема комнаты и свойства для получения состояния объекта.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 17

Описать класс, представляющий нелинейное уравнение вида $ax - \cos(x) = 0$. Описать метод, вычисляющий решение этого уравнения на заданном интервале методом деления пополам (см. раздел «Цикл с параметром for») и выбрасывающий исключение в случае отсутствия корня. Описать свойства для получения состояния объекта.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 18

Описать класс, представляющий квадратное уравнение вида $ax^2 + bx + c = 0$. Описать метод, вычисляющий решение этого уравнения и выбрасывающий исключение в случае отсутствия корней. Описать свойства для получения состояния объекта.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 19

Описать класс «процессор», содержащий сведения о марке, тактовой частоте, объеме кэша и стоимости. Предусмотреть инициализацию с проверкой допустимости значений полей. В случае недопустимых значений полей выбрасываются исключения. Описать свойства для получения состояния объекта.

Описать класс «материнская плата», включающий класс «процессор» и объем установленной оперативной памяти. Предусмотреть инициализацию с проверкой допустимости значений поля объема памяти. В случае недопустимых

значений поля выбрасывается исключение. Описать свойства для получения состояния объекта.

Написать программу, демонстрирующую все разработанные элементы классов.

Вариант 20

Описать класс «цветная точка». Для точки задаются координаты и цвет. Цвет описывается с помощью трех составляющих (красный, зеленый, синий). Предусмотреть различные методы инициализации объекта с проверкой допустимости значений. Допустимым диапазоном для каждой составляющей является [0, 255]. В случае недопустимых значений полей выбрасываются исключения. Описать свойства для получения состояния объекта и метод изменения цвета.

Написать программу, демонстрирующую все разработанные элементы класса.

Лабораторная работа 5. Одномерные массивы

Теоретический материал: глава 6, раздел «Массивы».

Вариант 1

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- сумму отрицательных элементов массива;
- произведение элементов массива, расположенных между максимальным и минимальным элементами.

Упорядочить элементы массива по возрастанию.

Вариант 2

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- сумму положительных элементов массива;
- произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.

Упорядочить элементы массива по убыванию.

Вариант 3

В одномерном массиве, состоящем из n целочисленных элементов, вычислить:

- произведение элементов массива с четными номерами;
- сумму элементов массива, расположенных между первым и последним нулевыми элементами.

Преобразовать массив таким образом, чтобы сначала располагались все положительные элементы, а потом — все отрицательные (элементы, равные нулю, считать положительными).

Вариант 4

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- сумму элементов массива с нечетными номерами;
- сумму элементов массива, расположенных между первым и последним отрицательными элементами.

Сжать массив, удалив из него все элементы, модуль которых не превышает единицу. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 5

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- максимальный элемент массива;
- сумму элементов массива, расположенных до последнего положительного элемента.

Сжать массив, удалив из него все элементы, модуль которых находится в интервале $[a, b]$. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 6

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- минимальный элемент массива;
- сумму элементов массива, расположенных между первым и последним положительными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, равные нулю, а потом — все остальные.

Вариант 7

В одномерном массиве, состоящем из n целочисленных элементов, вычислить:

- номер максимального элемента массива;
- произведение элементов массива, расположенных между первым и вторым нулевыми элементами.

Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в нечетных позициях, а во второй половине — элементы, стоявшие в четных позициях.

Вариант 8

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- номер минимального элемента массива;
- сумму элементов массива, расположенных между первым и вторым отрицательными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, модуль которых не превышает единицу, а потом — все остальные.

Вариант 9

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- максимальный по модулю элемент массива;
- сумму элементов массива, расположенных между первым и вторым положительными элементами.

Преобразовать массив таким образом, чтобы элементы, равные нулю, располагались после всех остальных.

Вариант 10

В одномерном массиве, состоящем из n целочисленных элементов, вычислить:

- минимальный по модулю элемент массива;
- сумму модулей элементов массива, расположенных после первого элемента, равного нулю.

Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в четных позициях, а во второй половине — элементы, стоявшие в нечетных позициях.

Вариант 11

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- номер минимального по модулю элемента массива;
- сумму модулей элементов массива, расположенных после первого отрицательного элемента.

Сжать массив, удалив из него все элементы, величина которых находится в интервале $[a, b]$. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 12

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- номер максимального по модулю элемента массива;
- сумму элементов массива, расположенных после первого положительного элемента.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых лежит в интервале $[a, b]$, а потом — все остальные.

Вариант 13

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- количество элементов массива, лежащих в диапазоне от A до B;
- сумму элементов массива, расположенных после максимального элемента.

Упорядочить элементы массива по убыванию модулей.

Вариант 14

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- количество элементов массива, равных нулю;
- сумму элементов массива, расположенных после минимального элемента.

Упорядочить элементы массива по возрастанию модулей.

Вариант 15

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- количество элементов массива, больших C;
- произведение элементов массива, расположенных после максимального по модулю элемента.

Преобразовать массив таким образом, чтобы сначала располагались все отрицательные элементы, а потом — все положительные (элементы, равные нулю, считать положительными).

Вариант 16

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- количество отрицательных элементов массива;
- сумму модулей элементов массива, расположенных после минимального по модулю элемента.

Заменить все отрицательные элементы массива их квадратами и упорядочить элементы массива по возрастанию.

Вариант 17

В одномерном массиве, состоящем из n целочисленных элементов, вычислить:

- количество положительных элементов массива;
- сумму элементов массива, расположенных после последнего элемента, равного нулю.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых не превышает единицу, а потом — все остальные.

Вариант 18

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- количество элементов массива, меньших C;
- сумму целых частей элементов массива, расположенных после последнего отрицательного элемента.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, отличающиеся от максимального не более чем на 20%, а потом — все остальные.

Вариант 19

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- произведение отрицательных элементов массива;
- сумму положительных элементов массива, расположенных до максимального элемента.

Изменить порядок следования элементов в массиве на обратный.

Вариант 20

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- произведение положительных элементов массива;
- сумму элементов массива, расположенных до минимального элемента.

Упорядочить по возрастанию отдельно элементы, стоящие на четных местах, и элементы, стоящие на нечетных местах.

Лабораторная работа 6. Двумерные массивы

Теоретический материал: глава 6, раздел «Массивы».

Вариант 1

Дана целочисленная прямоугольная матрица. Определить:

- количество строк, не содержащих ни одного нулевого элемента;
- максимальное из чисел, встречающихся в заданной матрице более одного раза.

Вариант 2

Дана целочисленная прямоугольная матрица. Определить количество столбцов, не содержащих ни одного нулевого элемента.

Характеристикой строки целочисленной матрицы назовем сумму ее положительных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с ростом характеристик.

Вариант 3

Дана целочисленная прямоугольная матрица. Определить:

- количество столбцов, содержащих хотя бы один нулевой элемент;
- номер строки, в которой находится самая длинная серия одинаковых элементов.

Вариант 4

Дана целочисленная квадратная матрица. Определить:

- произведение элементов в тех строках, которые не содержат отрицательных элементов;
- максимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

Вариант 5

Дана целочисленная квадратная матрица. Определить:

- сумму элементов в тех столбцах, которые не содержат отрицательных элементов;
- минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали матрицы.

Вариант 6

Дана целочисленная прямоугольная матрица. Определить:

- сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент;
- номера строк и столбцов всех седловых точек матрицы.

ПРИМЕЧАНИЕ

Матрица A имеет седловую точку A_{ij}, если A_{ij} является минимальным элементом в i-й строке и максимальным — в j-м столбце.

Вариант 7

Для заданной матрицы размером 8×8 найти такие k, при которых k-я строка матрицы совпадает с k-м столбцом.

Найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

Вариант 8

Характеристикой столбца целочисленной матрицы назовем сумму модулей его отрицательных нечетных элементов. Переставляя столбцы заданной матрицы, расположить их в соответствии с ростом характеристик.

Найти сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.

Вариант 9

Соседями элемента A_{ij} в матрице назовем элементы A_{kl}, где $i - 1 \le k \le i + 1$, $j - 1 \le l \le j + 1$, $(k, l) \ne (i, j)$. Операция сглаживания матрицы дает новую матрицу того же размера, каждый элемент которой получается как среднее арифметическое имеющихся соседей соответствующего элемента исходной матрицы. Построить результат сглаживания заданной вещественной матрицы размером 10 × 10.

В сглаженной матрице найти сумму модулей элементов, расположенных ниже главной диагонали.

Вариант 10

Элемент матрицы называется локальным минимумом, если он строго меньше всех имеющихся у него соседей (определение соседних элементов см. в варианте 9). Подсчитать количество локальных минимумов заданной матрицы размером 10 × 10.

Найти сумму модулей элементов, расположенных выше главной диагонали.

Вариант 11

Коэффициенты системы линейных уравнений заданы в виде прямоугольной матрицы. С помощью допустимых преобразований привести систему к треугольному виду.

Найти количество строк, среднее арифметическое элементов которых меньше заданной величины.

Вариант 12

Уплотнить заданную матрицу, удаляя из нее строки и столбцы, заполненные нулями.

Найти номер первой из строк, содержащих хотя бы один положительный элемент.

Вариант 13

Осуществить циклический сдвиг элементов прямоугольной матрицы на n элементов вправо или вниз (в зависимости от введенного режима). n может быть больше количества элементов в строке или столбце.

Вариант 14

Осуществить циклический сдвиг элементов квадратной матрицы размером $M \times N$ вправо на k элементов таким образом: элементы первой строки сдвигаются в последний столбец сверху вниз, из него — в последнюю строку справа налево, из нее — в первый столбец снизу вверх, из него — в первую строку; для остальных элементов — аналогично.

Вариант 15

Дана целочисленная прямоугольная матрица. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.

Характеристикой строки целочисленной матрицы назовем сумму ее отрицательных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с убыванием характеристик.

Вариант 16

Упорядочить строки целочисленной прямоугольной матрицы по возрастанию количества одинаковых элементов в каждой строке.

Найти номер первого из столбцов, не содержащих ни одного отрицательного элемента.

Вариант 17

Путем перестановки элементов квадратной вещественной матрицы добиться того, чтобы ее максимальный элемент находился в левом верхнем углу, следующий по величине — в позиции (2, 2), следующий по величине — в позиции (3, 3) и т. д., заполнив таким образом всю главную диагональ.

Найти номер первой из строк, не содержащих ни одного положительного элемента.

Вариант 18

Дана целочисленная прямоугольная матрица. Определить:

- количество строк, содержащих хотя бы один нулевой элемент;
- номер столбца, в котором находится самая длинная серия одинаковых элементов.

Вариант 19

Дана целочисленная квадратная матрица. Определить:

- сумму элементов в тех строках, которые не содержат отрицательных элементов;
- минимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

Вариант 20

Дана целочисленная прямоугольная матрица. Определить:

- количество отрицательных элементов в тех строках, которые содержат хотя бы один нулевой элемент;
- номера строк и столбцов всех седловых точек матрицы.

ПРИМЕЧАНИЕ

Матрица A имеет седловую точку A_{ij}, если A_{ij} является минимальным элементом в i-й строке и максимальным — в j-м столбце.

Лабораторная работа 7. Строки

Теоретический материал: глава 6, раздел «Символы и строки».

Вариант 1

Написать программу, которая считывает из текстового файла три предложения и выводит их в обратном порядке.

Вариант 2

Написать программу, которая считывает текст из файла и выводит на экран только предложения, содержащие введенное с клавиатуры слово.

Вариант 3

Написать программу, которая считывает текст из файла и выводит на экран только строки, содержащие двузначные числа.

Вариант 4

Написать программу, которая считывает английский текст из файла и выводит на экран слова, начинающиеся с гласных букв.

Вариант 5

Написать программу, которая считывает текст из файла и выводит его на экран, меняя местами каждые два соседних слова.

Вариант 6

Написать программу, которая считывает текст из файла и выводит на экран только предложения, не содержащие запятых.

Вариант 7

Написать программу, которая считывает текст из файла и определяет, сколько в нем слов, состоящих не более чем из четырех букв.

Вариант 8

Написать программу, которая считывает текст из файла и выводит на экран только цитаты, то есть предложения, заключенные в кавычки.

Вариант 9

Написать программу, которая считывает текст из файла и выводит на экран только предложения, состоящие из заданного количества слов.

Вариант 10

Написать программу, которая считывает английский текст из файла и выводит на экран слова текста, начинающиеся и оканчивающиеся на гласные буквы.

Вариант 11

Написать программу, которая считывает текст из файла и выводит на экран только строки, не содержащие двузначных чисел.

Вариант 12

Написать программу, которая считывает текст из файла и выводит на экран только предложения, начинающиеся с тире, перед которым могут находиться только пробельные символы.

Вариант 13

Написать программу, которая считывает английский текст из файла и выводит его на экран, заменив прописной каждую первую букву слов, начинающихся с гласной буквы.

Вариант 14

Написать программу, которая считывает текст из файла и выводит его на экран, заменив цифры от 0 до 9 словами «ноль», «один», ..., «девять», начиная каждое предложение с новой строки.

Вариант 15

Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в тексте.

Вариант 16

Написать программу, которая считывает текст из файла и выводит на экран сначала вопросительные, а затем восклицательные предложения.

Вариант 17

Написать программу, которая считывает текст из файла и выводит его на экран, после каждого предложения добавляя, сколько раз встретилось в нем введенное с клавиатуры слово.

Вариант 18

Написать программу, которая считывает текст из файла и выводит на экран все его предложения в обратном порядке.

Вариант 19

Написать программу, которая считывает текст из файла и выводит на экран сначала предложения, начинающиеся с однобуквенных слов, а затем все остальные.

Вариант 20

Написать программу, которая считывает текст из файла и выводит на экран предложения, содержащие максимальное количество знаков пунктуации.

Лабораторная работа 8. Классы и операции

Теоретический материал: глава 7.

Каждый разрабатываемый класс должен, как правило, содержать следующие элементы: скрытые поля, конструкторы с параметрами и без параметров, методы; свойства, индексаторы; перегруженные операции. Функциональные элементы класса должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. При возникновении ошибок должны выбрасываться исключения.

В программе должна выполняться проверка всех разработанных элементов класса.

Вариант 1

Описать класс для работы с одномерным массивом целых чисел (вектором). Обеспечить следующие возможности:

- задание произвольных целых границ индексов при создании объекта;
- обращение к отдельному элементу массива с контролем выхода за пределы массива;
- выполнение операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов;
- выполнение операций умножения и деления всех элементов массива на скаляр;
- вывод на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 2

Описать класс для работы с одномерным массивом строк фиксированной длины. Обеспечить следующие возможности:

- задание произвольных целых границ индексов при создании объекта;
- обращение к отдельной строке массива по индексу с контролем выхода за пределы массива;
- выполнение операций поэлементного сцепления двух массивов с образованием нового массива;
- выполнение операций слияния двух массивов с исключением повторяющихся элементов;
- вывод на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 3

Описать класс многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Обеспечить следующие возможности:

- вычисление значения многочлена для заданного аргумента;
- операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена;
- получение коэффициента, заданного по индексу;
- вывод на экран описания многочлена.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 4

Описать класс, обеспечивающий представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывода на экран подматрицы любого размера и всей матрицы, доступа по индексам к элементу матрицы.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 5

Описать класс для работы с восьмеричным числом, хранящимся в виде строки символов. Реализовать конструкторы, свойства, методы и следующие операции:

- операции присваивания, реализующие значимую семантику;
- операции сравнения;
- преобразование в десятичное число;
- форматный вывод;
- доступ к заданной цифре числа по индексу.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 6

Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (по авто-

ру, по году издания или категории), добавления книг в библиотеку, удаления книг из нее, доступа к книге по номеру.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 7

Описать класс «записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по фамилии и доступа к записи по номеру.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 8

Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, имени, дате рождения), добавления и удаления записей, сортировки по разным полям, доступа к записи по номеру.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 9

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- сложение, вычитание (как с другой матрицей, так и с числом);
- комбинированные операции присваивания (+=, -=);
- операции сравнения на равенство/неравенство;
- операции вычисления обратной и транспонированной матрицы;
- доступ к элементу по индексам.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 10

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- умножение, деление (как на другую матрицу, так и на число);
- комбинированные операции присваивания (*=, /=);
- операцию возведения в степень;
- методы вычисления детерминанта и нормы;
- доступ к элементу по индексам.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 11

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- методы, реализующие проверку типа матрицы (квадратная, диагональная, нулевая, единичная, симметричная, верхняя треугольная, нижняя треугольная);
- операции сравнения на равенство/неравенство;
- доступ к элементу по индексам.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 12

Описать класс «множество», позволяющий выполнять основные операции: добавление и удаление элемента, пересечение, объединение и разность множеств.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 13

Описать класс «предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Предусмотреть возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 14

Описать класс «автостоянка» для хранения сведений об автомобилях. Для каждого автомобиля записываются госномер, цвет, фамилия владельца и признак присутствия на стоянке. Обеспечить возможность поиска автомобиля по разным критериям, вывода списка присутствующих и отсутствующих на стоянке автомобилей, доступа к имеющимся сведениям по номеру места.

Написать программу, демонстрирующую все разработанные элементы класса.

Вариант 15

Описать класс «колода карт», включающий закрытый массив элементов класса «карта». В карте хранятся масть и номер. Обеспечить возможность вывода карты по номеру, вывода всех карт, перемешивания колоды и выдачи всех карт из колоды поодиночке и по 6 штук в случайном порядке.

Написать программу, демонстрирующую все разработанные элементы классов.

Вариант 16

Описать класс «поезд», содержащий следующие закрытые поля:

- название пункта назначения;

- номер поезда (может содержать буквы и цифры);
- время отправления.

Предусмотреть свойства для получения состояния объекта.

Описать класс «вокзал», содержащий закрытый массив поездов. Обеспечить следующие возможности:

- вывод информации о поезде по номеру с помощью индекса;
- вывод информации о поездах, отправляющихся после введенного с клавиатуры времени;
- перегруженную операцию сравнения, выполняющую сравнение времени отправления двух поездов;
- вывод информации о поездах, отправляющихся в заданный пункт назначения.

Информация должна быть отсортирована по времени отправления. Написать программу, демонстрирующую все разработанные элементы классов.

Вариант 17

Описать класс «товар», содержащий следующие закрытые поля:

- название товара;
- название магазина, в котором продается товар;
- стоимость товара в рублях.

Предусмотреть свойства для получения состояния объекта.

Описать класс «склад», содержащий закрытый массив товаров. Обеспечить следующие возможности:

- вывод информации о товаре по номеру с помощью индекса;
- вывод на экран информации о товаре, название которого введено с клавиатуры; если таких товаров нет, выдать соответствующее сообщение;
- сортировку товаров по названию магазина, по наименованию и по цене;
- перегруженную операцию сложения товаров, выполняющую сложение их цен.

Написать программу, демонстрирующую все разработанные элементы классов.

Вариант 18

Описать класс «самолет», содержащий следующие закрытые поля:

- название пункта назначения;
- шестизначный номер рейса;
- время отправления.

Предусмотреть свойства для получения состояния объекта.

Описать класс «аэропорт», содержащий закрытый массив самолетов. Обеспечить следующие возможности:

- вывод информации о самолете по номеру рейса с помощью индекса;

- вывод информации о самолетах, отправляющихся в течение часа после введенного с клавиатуры времени;
- вывод информации о самолетах, отправляющихся в заданный пункт назначения;
- перегруженную операцию сравнения, выполняющую сравнение времени отправления двух самолетов.

Информация должна быть отсортирована по времени отправления. Написать программу, демонстрирующую все разработанные элементы классов.

Вариант 19

Описать класс «запись», содержащий следующие закрытые поля:

- фамилия, имя;
- номер телефона;
- дата рождения (массив из трех чисел).

Предусмотреть свойства для получения состояния объекта.

Описать класс «записная книжка», содержащий закрытый массив записей. Обеспечить следующие возможности:

- вывод на экран информации о человеке, номер телефона которого введен с клавиатуры; если такого нет, выдать на дисплей соответствующее сообщение;
- поиск людей, день рождения которых сегодня или в заданный день;
- поиск людей, день рождения которых будет на следующей неделе;
- поиск людей, номер телефона которых начинается на три заданных цифры.

Написать программу, демонстрирующую все разработанные элементы классов.

Вариант 20

Описать класс «англо-русский словарь», обеспечивающий возможность хранения нескольких вариантов перевода для каждого слова. Реализовать доступ по строковому индексу — английскому слову. Обеспечить возможность вывода всех значений слов по заданному префиксу.

Лабораторная работа 9. Наследование

Теоретический материал: глава 8.

В программах требуется описать базовый класс (возможно, абстрактный), в котором с помощью виртуальных или абстрактных методов и свойств задается интерфейс для производных классов. Целью лабораторной работы является максимальное использование наследования, даже если для конкретной задачи оно не дает

выигрыша в объеме программы. Во всех классах следует переопределить метод Equals, чтобы обеспечить сравнение значений, а не ссылок.

Функция Main должна содержать массив из элементов базового класса, заполненный ссылками на производные классы. В этой функции должно демонстрироваться использование всех разработанных элементов классов.

Вариант 1

Создать класс Point (точка). На его основе создать классы ColoredPoint и Line (линия). На основе класса Line создать классы ColoredLine и PolyLine (многоугольник). В классах описать следующие элементы:

- конструкторы с параметрами и конструкторы по умолчанию;
- свойства для установки и получения значений всех координат, а также для изменения цвета и получения текущего цвета;
- для линий — методы изменения угла поворота линий относительно первой точки;
- для многоугольника — метод масштабирования.

Вариант 2

Создать абстрактный класс Vehicle (транспортное средство). На его основе реализовать классы Plane (самолет), Car (автомобиль) и Ship (корабль). Классы должны иметь возможность задавать и получать координаты и параметры средств передвижения (цена, скорость, год выпуска и т. п.) с помощью свойств. Для самолета должна быть определена высота, для самолета и корабля — количество пассажиров, для корабля — порт приписки. Динамические характеристики задать с помощью методов.

Вариант 3

Описать базовый класс Строка.

Обязательные поля класса:

- поле для хранения символов строки;
- значение типа word для хранения длины строки в байтах.

Реализовать обязательные методы следующего назначения:

- конструктор без параметров;
- конструктор, принимающий в качестве параметра строковый литерал;
- конструктор, принимающий в качестве параметра символ;
- метод получения длины строки;
- метод очистки строки (сделать строку пустой).

Описать производный от Строка класс Комплексное_число.

Строки данного класса состоят из двух полей, разделенных символом i.

Первое поле задает значение действительной части числа, второе — значение мнимой. Каждое из полей может содержать только символы десятичных цифр и символы – и +, задающие знак числа. Символы – или + могут находиться только в первой позиции числа, причем символ + может отсутствовать, в этом случае число считается положительным. Если в составе инициализирующей строки будут встречены любые символы, отличные от допустимых, класс Комплексное_число принимает нулевое значение. Примеры строк:

```
33i12
-7i100
+5i - 21
```

Для класса Комплексное_число определить следующие методы:

- проверка на равенство;
- сложение чисел;
- умножение чисел.

Вариант 4

Описать базовый класс Строка в соответствии с вариантом 3.

Описать производный от Строка класс Десятичная_строка.

Строки данного класса могут содержать только символы десятичных цифр и символы – и +, задающие знак числа. Символы – или + могут находиться только в первой позиции числа, причем символ + может отсутствовать, в этом случае число считается положительным. Если в составе инициализирующей строки будут встречены любые символы, отличные от допустимых, класс Десятичная_строка принимает нулевое значение. Содержимое данных строк рассматривается как десятичное число.

Для класса определить следующие методы:

- конструктор, принимающий в качестве параметра число;
- арифметическая разность строк;
- проверка на больше (по значению);
- проверка на меньше (по значению).

Вариант 5

Описать базовый класс Строка в соответствии с вариантом 3.

Описать производный от Строка класс Битовая_строка.

Строки данного класса могут содержать только символы '0' или '1'. Если в составе инициализирующей строки будут встречены любые символы, отличные от допустимых, класс Битовая_строка принимает нулевое значение. Содержимое данных строк рассматривается как двоичное число. Отрицательные числа хранятся в дополнительном коде.

Для класса Битовая_строка определить следующие методы:

- конструктор, принимающий в качестве параметра строковый литерал;
- деструктор;
- изменение знака на противоположный (перевод числа в дополнительный код);
- присваивание;
- вычисление арифметической суммы строк;
- проверка на равенство.

В случае необходимости более короткая битовая строка расширяется влево знаковым разрядом.

Вариант 6

1. Описать базовый класс Элемент.

   Закрытые поля:

   - имя элемента (строка символов);
   - количество входов элемента;
   - количество выходов элемента.

   Методы:

   - конструктор класса без параметров;
   - конструктор, задающий имя и устанавливающий равным 1 количество входов и выходов;
   - конструктор, задающий значения всех полей элемента.

   Свойства:

   - имя элемента (только чтение);
   - количество входов элемента;
   - количество выходов элемента.

2. На основе класса Элемент описать производный класс Комбинационный, представляющий собой комбинационный элемент (двоичный вентиль), который может иметь несколько входов и один выход.

   Поле — массив значений входов.

   Методы:

   - конструкторы;
   - метод, задающий значение на входах экземпляра класса;
   - метод, позволяющий опрашивать состояние отдельного входа экземпляра класса;
   - метод, вычисляющий значение выхода (по варианту задания).

3. На основе класса Элемент описать производный класс Память, представляющий собой триггер. Триггер имеет входы, соответствующие типу триггера

(см. далее вариант задания), и входы установки и сброса. Все триггеры считаются синхронными, сам синхровход в состав триггера не включается.

Поля:

- массив значений входов объекта класса, в массиве учитываются все входы (управляющие и информационные);
- состояние на прямом выходе триггера;
- состояние на инверсном выходе триггера.

Методы:

- конструктор (по умолчанию сбрасывает экземпляр класса);
- конструктор копирования;
- метод, задающий значение на входах экземпляра класса;
- методы, позволяющие опрашивать состояния отдельного входа экземпляра класса;
- метод, вычисляющий состояние экземпляра класса (по варианту задания) в зависимости от текущего состояния и значений на входах;
- метод, переопределяющий операцию == для экземпляров класса.

4. Создать класс `Регистр`, используя класс `Память` как вложенный класс.

   Поля:

   - состояние входа «Сброс» — один для экземпляра класса;
   - состояние входа «Установка» — один для экземпляра класса;
   - массив типа `Память` заданной в варианте размерности;
   - массив (массивы), содержащий значения на соответствующих входах элементов массива типа `Память`.

   Методы:

   - метод, задающий значение на входах экземпляра класса;
   - метод, позволяющий опрашивать состояние отдельного выхода экземпляра класса;
   - метод, вычисляющий значение нового состояния экземпляра класса.

   Все поля классов `Элемент`, `Комбинационный` и `Память` должны быть описаны с ключевым словом `private`.

   В задании перечислены только обязательные члены и методы класса. Можно задавать дополнительные члены и методы, если они не отменяют обязательные и обеспечивают дополнительные удобства при работе с данными классами, например, описать функции вычисления выхода/состояния как виртуальные.

5. Для проверки функционирования созданных классов написать программу, использующую эти классы. В программе должны быть продемонстрированы все свойства созданных классов.

Конкретный тип комбинационного элемента, тип триггера и разрядность регистра выбираются в соответствии с вариантом задания:

Вариант	Комбинационный элемент	Число входов	Триггер	Разрядность регистра
1	И-НЕ	4	RS	8
2	ИЛИ	5	RST	10
3	МОД2-НЕ	6	D	12
4	И	8	T	8
5	ИЛИ-НЕ	8	V	9
6	И	4	RS	10
7	ИЛИ-НЕ	5	JK	11
8	МОД2	5	D	8
9	И	4	T	10
10	ИЛИ	3	JK	8
11	И-НЕ	3	RS	12
12	ИЛИ-НЕ	4	RST	4
13	МОД2	5	D	10
14	МОД2-НЕ	6	T	10
15	ИЛИ-НЕ	8	V	10
16	И	8	JK	6
17	И-НЕ	8	RS	10
18	ИЛИ	8	T	10
19	МОД2	6	JK	8
20	МОД2-НЕ	5	V	10

Лабораторная работа 10. Структуры

Теоретический материал: глава 9.

Вариант 1

Описать структуру с именем STUDENT, содержащую следующие поля:

- фамилия и инициалы;
- номер группы;
- успеваемость (массив из пяти элементов).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT (записи должны быть упорядочены по возрастанию номера группы);
- вывод на экран фамилий и номеров групп для всех студентов, включенных в массив, если средний балл студента больше 4,0 (если таких студентов нет, вывести соответствующее сообщение).

Вариант 2

Описать структуру с именем STUDENT, содержащую следующие поля:

- фамилия и инициалы;
- номер группы;
- успеваемость (массив из пяти элементов).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT (записи должны быть упорядочены по возрастанию среднего балла);
- вывод на экран фамилий и номеров групп для всех студентов, имеющих оценки 4 и 5 (если таких студентов нет, вывести соответствующее сообщение).

Вариант 3

Описать структуру с именем STUDENT, содержащую следующие поля:

- фамилия и инициалы;
- номер группы;
- успеваемость (массив из пяти элементов).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT (записи должны быть упорядочены по алфавиту);
- вывод на экран фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2 (если таких студентов нет, вывести соответствующее сообщение).

Вариант 4

Описать структуру с именем AEROFLOT, содержащую следующие поля:

- название пункта назначения рейса;
- номер рейса;
- тип самолета.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из семи элементов типа AEROFLOT (записи должны быть упорядочены по возрастанию номера рейса);
- вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры (если таких рейсов нет, вывести соответствующее сообщение).

Вариант 5

Описать структуру с именем AEROFLOT, содержащую следующие поля:

- название пункта назначения рейса;
- номер рейса;
- тип самолета.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из семи элементов типа AEROFLOT (записи должны быть размещены в алфавитном порядке по названиям пунктов назначения);
- вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры (если таких рейсов нет, вывести соответствующее сообщение).

Вариант 6

Описать структуру с именем WORKER, содержащую следующие поля:

- фамилия и инициалы работника;
- название занимаемой должности;
- год поступления на работу.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти структур типа WORKER (записи должны быть упорядочены по алфавиту);
- вывод на экран фамилий работников, стаж работы которых превышает значение, введенное с клавиатуры (если таких работников нет, вывести соответствующее сообщение).

Вариант 7

Описать структуру с именем TRAIN, содержащую следующие поля:

- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа TRAIN (записи должны быть размещены в алфавитном порядке по названиям пунктов назначения);
- вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени (если таких поездов нет, вывести соответствующее сообщение).

Вариант 8

Описать структуру с именем TRAIN, содержащую следующие поля:

- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из шести элементов типа TRAIN (записи должны быть упорядочены по времени отправления поезда);
- вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры (если таких поездов нет, вывести соответствующее сообщение).

Вариант 9

Описать структуру с именем TRAIN, содержащую следующие поля:

- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа TRAIN (записи должны быть упорядочены по номерам поездов);
- вывод на экран информации о поезде, номер которого введен с клавиатуры (если таких поездов нет, вывести соответствующее сообщение).

Вариант 10

Описать структуру с именем MARSH, содержащую следующие поля:

- название начального пункта маршрута;
- название конечного пункта маршрута;
- номер маршрута.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа MARSH (записи должны быть упорядочены по номерам маршрутов);
- вывод на экран информации о маршруте, номер которого введен с клавиатуры (если таких маршрутов нет, вывести соответствующее сообщение).

Вариант 11

Описать структуру с именем MARSH, содержащую следующие поля:

- название начального пункта маршрута;
- название конечного пункта маршрута;
- номер маршрута.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа MARSH (записи должны быть упорядочены по номерам маршрутов);
- вывод на экран информации о маршрутах, которые начинаются или оканчиваются в пункте, название которого введено с клавиатуры (если таких маршрутов нет, вывести соответствующее сообщение).

Вариант 12

Описать структуру с именем NOTE, содержащую следующие поля:

- фамилия, имя;
- номер телефона;
- дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE (записи должны быть упорядочены по дате рождения);
- вывод на экран информации о человеке, номер телефона которого введен с клавиатуры (если такого нет, вывести соответствующее сообщение).

Вариант 13

Описать структуру с именем NOTE, содержащую следующие поля:

- фамилия, имя;
- номер телефона;
- дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE (записи должны быть размещены по алфавиту);
- вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры (если таких нет, вывести соответствующее сообщение).

Вариант 14

Описать структуру с именем NOTE, содержащую следующие поля:

- фамилия, имя;
- номер телефона;
- дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE (записи должны быть упорядочены по трем первым цифрам номера телефона);
- вывод на экран информации о человеке, чья фамилия введена с клавиатуры (если такого нет, вывести соответствующее сообщение).

Вариант 15

Описать структуру с именем ZNAK, содержащую следующие поля:

- фамилия, имя;
- знак Зодиака;
- дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK (записи должны быть упорядочены по дате рождения);
- вывод на экран информации о человеке, чья фамилия введена с клавиатуры (если такого нет, вывести соответствующее сообщение).

Вариант 16

Описать структуру с именем ZNAK, содержащую следующие поля:

- фамилия, имя;
- знак Зодиака;
- дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK (записи должны быть упорядочены по дате рождения);
- вывод на экран информации о людях, родившихся под знаком, название которого введено с клавиатуры (если таких нет, вывести соответствующее сообщение).

Вариант 17

Описать структуру с именем ZNAK, содержащую следующие поля:

- фамилия, имя;
- знак Зодиака;
- дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK (записи должны быть упорядочены по знакам Зодиака);
- вывод на экран информации о людях, родившихся в месяц, значение которого введено с клавиатуры (если таких нет, вывести соответствующее сообщение).

Вариант 18

Описать структуру с именем PRICE, содержащую следующие поля:

- название товара;
- название магазина, в котором продается товар;
- стоимость товара в рублях.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа PRICE (записи должны быть упорядочены в алфавитном порядке по названиям товаров);

- вывод на экран информации о товаре, название которого введено с клавиатуры (если таких товаров нет, вывести соответствующее сообщение).

Вариант 19

Описать структуру с именем PRICE, содержащую следующие поля:

- название товара;
- название магазина, в котором продается товар;
- стоимость товара в рублях.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа PRICE (записи должны быть упорядочены в алфавитном порядке по названиям магазинов);
- вывод на экран информации о товарах, продающихся в магазине, название которого введено с клавиатуры (если такого магазина нет, вывести соответствующее сообщение).

Вариант 20

Описать структуру с именем ORDER, содержащую следующие поля:

- расчетный счет плательщика;
- расчетный счет получателя;
- перечисляемая сумма в рублях.

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ORDER (записи должны быть размещены в алфавитном порядке по расчетным счетам плательщиков);
- вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры (если такого расчетного счета нет, вывести соответствующее сообщение).

Лабораторная работа 11. Интерфейсы и параметризованные коллекции

Теоретический материал: главы 9, 13.

Выполнить задания лабораторной работы 9, используя для хранения экземпляров разработанных классов стандартные параметризованные коллекции. Во всех классах реализовать интерфейс IComparable и перегрузить операции отношения для реализации значимой семантики сравнения объектов по какому-либо полю на усмотрение студента.

Лабораторная работа 12. Создание Windows-приложений

Теоретический материал: глава 14.

Задание 1. Диалоговые окна

Общая часть задания: написать Windows-приложение, заголовок главного окна которого содержит Ф. И. О., группу и номер варианта. В программе должна быть предусмотрена обработка исключений, возникающих из-за ошибочного ввода пользователя

Вариант 1

Создать меню с командами Input, Calc и Exit.

При выборе команды Input открывается диалоговое окно, содержащее:

- три поля типа TextBox для ввода длин трех сторон треугольника;
- группу из двух флажков (Периметр и Площадь) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода длин трех сторон треугольника;
- выбора режима с помощью флажков: подсчет периметра и/или площади треугольника.

При выборе команды Calc открывается диалоговое окно с результатами. При выборе команды Exit приложение завершается.

Вариант 2

Создать меню с командами Size, Color, Paint, Quit.

Команда Paint недоступна. При выборе команды Quit приложение завершается. При выборе команды Size открывается диалоговое окно, содержащее:

- два поля типа TextBox для ввода длин сторон прямоугольника;
- группу из трех флажков (Red, Green, Blue) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода длин сторон прямоугольника в пикселах в поля ввода;
- выбора его цвета с помощью флажков.

После задания параметров команда Paint становится доступной.

При выборе команды Paint в главном окне приложения выводится прямоугольник заданного размера и сочетания цветов или выдается сообщение, если введенные размеры превышают размер окна.

Вариант 3

Создать меню с командами Input, Work, Exit.

При выборе команды Exit приложение завершает работу. При выборе команды Input открывается диалоговое окно, содержащее:

- три поля ввода типа TextBox с метками Radius, Height, Density;
- группу из двух флажков (Volume, Mass) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода радиуса, высоты и плотности конуса;
- выбора режима с помощью флажков: подсчет объема и/или массы конуса.

При выборе команды Work открывается окно сообщений с результатами.

Вариант 4

Создать меню с командами Input, Calc, Draw, Exit.

При выборе команды Exit приложение завершает работу. При выборе команды Input открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой Radius;
- группу из двух флажков (Square, Length) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода радиуса окружности;
- выбора режима с помощью флажков: подсчет площади круга (Square) и/или длины окружности (Length).

При выборе команды Calc открывается окно сообщений с результатами. При выборе команды Draw в центре главного окна выводится круг введенного радиуса или выдается сообщение, что рисование невозможно (если диаметр превышает размеры рабочей области).

Вариант 5

Создать меню с командами Input, Calc, About.

При выборе команды About открывается окно с информацией о разработчике. При выборе команды Input открывается диалоговое окно, содержащее:

- три поля ввода типа TextBox с метками Number 1, Number 2, Number 3;
- группу из двух флажков (Summ, Least multiple) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность ввода трех чисел и выбора режима вычислений с помощью флажков: подсчет суммы трех чисел (Summ) и/или наименьшего общего кратного двух первых чисел (Least multiple). При выборе команды Calc открывается диалоговое окно с результатами.

Вариант 6

Создать меню с командами Input, Calc, Quit.

Команда Calc недоступна. При выборе команды Quit приложение завершается. При выборе команды Input открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox с метками Number 1, Number 2;
- группу из трех флажков (Summa, Max divisor, Multiply) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода двух чисел;
- выбора режима вычислений с помощью флажков (можно вычислять в любой комбинации такие величины, как сумма, наибольший общий делитель и произведение двух чисел).

При выборе команды Calc открывается окно сообщений с результатами.

Вариант 7

Создать меню с командами Begin, Help, About.

При выборе команды About открывается окно с информацией о разработчике. При выборе команды Begin открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой input;
- метку типа Label для вывода результата;
- группу из трех переключателей (2, 8, 16) типа RadioButton;
- две кнопки типа Button — Do и OK.

Обеспечить возможность:

- ввода числа в десятичной системе в поле input;
- выбора режима преобразования с помощью переключателей: перевод в двоичную, восьмеричную или шестнадцатеричную систему счисления.

При щелчке на кнопке Do должен появляться результат перевода.

Вариант 8

Создать меню с командами Input color, Change, Exit, Help.

При выборе команды Exit приложение завершает работу. При выборе команды Input color открывается диалоговое окно, содержащее:

- три поля ввода типа TextBox с метками Red, Green, Blue;
- группу из двух флажков (Left, Right) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность ввода RGB-составляющих цвета. При выборе команды Change цвет главного окна изменяется на заданный (левая, правая или обе половины окна в зависимости от установки флажков).

Вариант 9

Создать меню с командами Input size, Choose, Change, Exit.

При выборе команды Exit приложение завершает работу. Команда Change недоступна. При выборе команды Input size открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox с метками Size x, Size y;
- кнопку типа Button.

При выборе команды Choose открывается диалоговое окно, содержащее:

- группу из двух переключателей (Increase, Decrease) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность ввода значений в поля Size x и Size y. Значения интерпретируются как количество пикселов, на которое надо изменить размеры главного окна (увеличить или уменьшить в зависимости от положения переключателей).

После ввода значений команда Change становится доступной. При выборе этой команды размеры главного окна увеличиваются или уменьшаются на введенное количество пикселов.

Вариант 10

Создать меню с командами Begin, Work, About.

При выборе команды About открывается окно с информацией о разработчике. При выборе команды Begin открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой Input word;
- группу из двух переключателей (Upper case, Lower case) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность ввода слова и выбора режима перевода в верхний или нижний регистр в зависимости от положения переключателей. При выборе команды Work открывается диалоговое окно с результатом перевода.

Вариант 11

Создать меню с командами Input color, Change, Clear.

При выборе команды Input color открывается диалоговое окно, содержащее:

- группу из двух флажков (Up, Down) типа CheckBox;
- группу из трех переключателей (Red, Green, Blue) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность:

- выбора цвета с помощью переключателей;
- ввода режима, определяющего, какая область закрашивается: все окно, его верхняя или нижняя половина.

При выборе команды Change цвет главного окна изменяется на заданный (верхняя, нижняя или обе половины в зависимости от введенного режима). При выборе команды Clear восстанавливается первоначальный цвет окна.

Вариант 12

Создать меню с командами Translate, Help, About, Exit.

При выборе команды Exit приложение завершает работу. При выборе команды Translate открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой Binary number;
- поле ввода типа TextBox для вывода результата (read-only);
- группу из трех переключателей (8, 10, 16) типа RadioButton;
- кнопку Do типа Button.

Обеспечить возможность:

- ввода числа в двоичной системе в поле Binary number;
- выбора режима преобразования с помощью переключателей: перевод в восьмеричную, десятичную или шестнадцатеричную систему счисления.

При щелчке на кнопке Do должен появляться результат перевода.

Вариант 13

Создать меню с командами Reverse, About, Exit.

При выборе команды About открывается окно с информацией о разработчике. При выборе команды Reverse открывается диалоговое окно, содержащее:

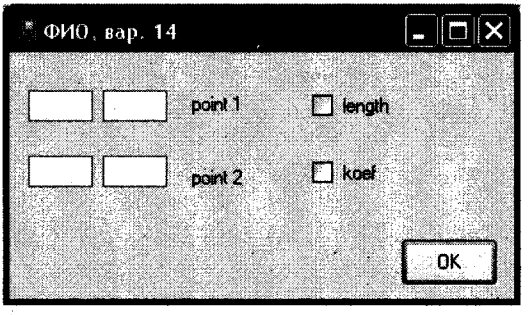
- поле ввода типа TextBox с меткой Input;
- группу из двух переключателей (Upper case, Reverse) типа CheckBox;
- кнопку OK типа Button.

Обеспечить возможность ввода фразы и выбора режима: перевод в верхний регистр и/или изменение порядка следования символов на обратный в зависимости от состояния переключателей. Результат преобразования выводится в исходное поле ввода.

Вариант 14

Создать меню с командами Input, Show и Exit.

При выборе команды Exit приложение завершает работу. При выборе команды Input открывается диалоговое окно вида:



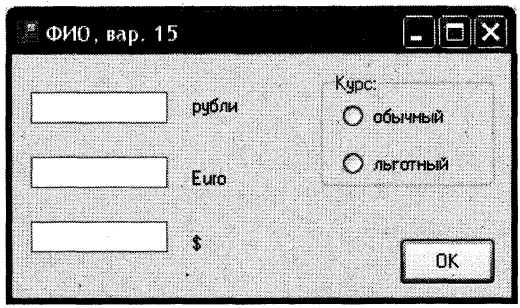
Обеспечивается возможность ввода координат двух точек и выбора режима с помощью флажков length и koef: подсчет длины отрезка, соединяющего эти точки, и/или углового коэффициента.

При выборе команды Show открывается окно сообщений с результатами подсчета.

Вариант 15

Создать меню с командами Input, About и Exit.

При выборе команды Exit приложение завершает работу. При выборе команды About открывается окно с информацией о разработчике. При выборе команды Input открывается диалоговое окно вида:



Обеспечивается возможность ввода суммы в рублях и перевода ее в евро и доллары по обычному или льготному курсу. Поля Euro и $ доступны только для чтения.

Вариант 16

Создать меню с командами Begin, Work, About.

При выборе команды About открывается окно с информацией о разработчике. При выборе команды Begin открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox;
- группу из двух переключателей (First letter, All letters) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность ввода предложения и выбора режима его преобразования: либо начинать с прописной буквы каждое слово (First letter), либо перевести все буквы в верхний регистр (All letters). При выборе команды Work открывается диалоговое окно с результатом преобразования.

Вариант 17

Написать анализатор текстовых файлов, выводящий информацию о количестве слов в тексте, а также статистическую информацию о введенной пользователем букве.

Создать следующую систему меню:

- Файл
  - Загрузить текст
  - Выход
- Анализ
  - Количество слов
  - Повторяемость буквы

При выборе файла для загрузки использовать объект типа `OpenFileDialog`. При выборе команды **Количество слов** программа должна вывести в окно сообщений количество слов в тексте.

При выборе команды **Повторяемость буквы** программа предлагает пользователю ввести букву, а затем выводит количество ее повторений без учета регистра в окно сообщений.

Вариант 18

Создать редактор текстовых файлов с возможностью сохранения текста в формате HTML. Создать следующую систему меню:

- Файл
  - Загрузить текст
  - Сохранить как текст
  - Сохранить как HTML
- Выход

При выборе файла для загрузки использовать объект `OpenFileDialog`. При выборе файла для сохранения использовать объект `SaveFileDialog`. Для редактирования текста использовать объект `Memo`.

При сохранении текста в формате HTML текст записывать в файл с заменой:

- всех пробелов на символы ` `;
- всех символов перевода строки на символы `<BR>`;
- всех символов < на символы `<`;
- всех символов > на символы `>`;
- всех символов & на символы `&`;
- всех символов " (двойные кавычки) на символы `&quot;`.

Вариант 19

Создать меню с командами **Input**, **Draw**, **Clear**.

При выборе команды **Input** открывается диалоговое окно, содержащее:

- четыре поля для ввода координат двух точек;
- группу из трех переключателей (**Red**, **Green**, **Blue**) типа `RadioButton`;
- кнопку типа `Button`.

При выборе команды Draw в главное окно выводится отрезок прямой выбранного цвета с координатами концов отрезка, заданными в диалоговом окне. При выборе команды Clear отрезок стирается.

Вариант 20

Создать меню с командами Input, Change, Exit.

При выборе команды Exit приложение завершает работу. Команда Change недоступна. В центре главного окна выведен квадрат размером 100 × 100 пикселов. При выборе команды Input открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox с метками Size x, Size y;
- группу из двух переключателей (Increase, Decrease) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность ввода значений в поля Size x и Size y. Значения интерпретируются как количество пикселов, на которое надо изменить размеры квадрата, выведенного в главное окно (увеличить или уменьшить в зависимости от положения переключателей).

После ввода значений команда Change становится доступной. При выборе этой команды размеры квадрата увеличиваются или уменьшаются на введенное количество пикселов. Если квадрат выходит за пределы рабочей области окна, выдается сообщение.

Вариант 21

Написать Windows-приложение, которое по заданным в файле исходным данным выводит информацию о компьютерах.

Создать меню с командами Choose, Show, Quit.

Команда Show недоступна. Команда Quit завершает работу приложения.

При запуске приложения из файла читаются исходные данные. Файл необходимо сформировать самостоятельно. Каждая строка файла содержит тип компьютера, цену (price) и емкость жесткого диска (hard drive).

При выборе команды Choose открывается диалоговое окно, содержащее:

- поле типа TextBox для ввода минимальной емкости диска;
- поле типа TextBox для ввода максимальной приемлемой цены;
- группу из двух переключателей (Hard drive, Price) типа RadioButton;
- OK, Cancel — кнопки типа Button.

После ввода всех данных команда меню Show становится доступной. Команда Show открывает диалоговое окно, содержащее список компьютеров, удовлетворяющий введенным ограничениям и упорядоченный по отмеченной характеристике.

Задание 2. Структуры и параметризованные коллекции

Описать структуру, соответствующую заданиям лабораторной работы 10. Создать параметризованную коллекцию (см. раздел «Классы-прототипы») для хранения описанной структуры. Вид коллекции выбрать самостоятельно. Написать Windows-приложение для работы с этой коллекцией, позволяющее выполнять:

1. добавление элемента в коллекцию с клавиатуры;
2. считывание данных из файла;
3. запись данных в тот же или указанный файл;
4. сортировку данных по различным критериям;
5. поиск элемента по заданному полю;
6. вывод всех элементов, удовлетворяющих заданному условию;
7. удаление элемента из коллекции[1].

Приложение должно содержать меню и диалоговые окна и предусматривать обработку возможных ошибок пользователя с помощью исключений.

Задание 3. Графика в Windows

Вариант 1

Написать Windows-приложение, которое выполняет анимацию изображения.

Создать меню с командами Show picture, Choose, Animate, Stop, Quit.

Команда Quit завершает работу приложения. При выборе команды Show picture в центре экрана рисуется объект, состоящий из нескольких графических примитивов.

При выборе команды Choose открывается диалоговое окно, содержащее:

- поле типа `TextBox` с меткой Speed для ввода скорости движения объекта;
- группу Direction из двух переключателей (Up-Down, Left-Right) типа `RadioButton` для выбора направления движения;
- кнопку типа `Button`.

По команде Animate объект начинает перемещаться в выбранном направлении до края окна и обратно с заданной скоростью, по команде Stop — прекращает движение.

Вариант 2

Написать Windows-приложение, которое по заданным в файле исходным данным строит график или столбиковую диаграмму.

Создать меню с командами Input data, Choose, Line, Bar, Quit.

---

[1] Номера пунктов для каждого варианта задаются преподавателем. Например, для варианта 1 — пункты 1, 2, 4 (сортировка по возрастанию номера группы), 6 (вывод списка всех студентов, средний балл которых больше 4,0).

Команды Line и Bar недоступны. Команда Quit завершает работу приложения.

При выборе команды Input data из файла читаются исходные данные (файл сформировать самостоятельно).

По команде Choose открывается диалоговое окно, содержащее:

- список для выбора цвета графика типа ListBox;
- группу из двух переключателей (Line, Bar) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность ввода цвета и выбора режима: построение графика (Line) или столбиковой диаграммы (Bar). После указания параметров становится доступной соответствующая команда меню.

По команде Line или Bar в главном окне приложения выбранным цветом строится график или диаграмма. Окно должно содержать заголовок графика или диаграммы, наименование и градацию осей. Изображение должно занимать все окно и масштабироваться при изменении размеров окна.

Вариант 3

Написать Windows-приложение, которое строит графики четырех заданных функций.

Создать меню с командами Chart, Build, Clear, About, Quit.

Команда Quit завершает работу приложения. При выборе команды About открывается окно с информацией о разработчике.

Команда Chart открывает диалоговое окно, содержащее:

- список для выбора цвета графика типа ListBox;
- список для выбора типа графика типа ListBox, содержащий четыре пункта: sin(x), sin(x+π/4), cos(x), cos(x–π/4);
- кнопку типа Button.

Обеспечить возможность выбора цвета и вида графика. После щелчка на кнопке OK в главном окне приложения строится график выбранной функции на интервале от $-\pi/2$ до $+\pi/2$. Окно должно содержать заголовок графика, наименование и градацию осей. Изображение должно занимать все окно и масштабироваться при изменении размеров окна.

Команда Clear очищает окно.

Вариант 4

Написать Windows-приложение — графическую иллюстрацию сортировки методом выбора.

Создать меню с командами File, Animate, About, Exit.

Команда Animate недоступна. Команда Exit завершает работу приложения. Команда About открывает окно с информацией о разработчике. Для выбора файла исходных данных (команда File) использовать объект класса OpenFileDialog.

Из выбранного файла читаются исходные данные для сортировки (сформировать самостоятельно не менее трех файлов различной длины с данными целого типа).

После чтения данных становится доступной команда Animate.

При выборе команды Animate в главном окне приложения отображается процесс сортировки в виде столбиковой диаграммы. Каждый элемент представляется столбиком соответствующего размера. На каждом шаге алгоритма два элемента меняются местами. Окно должно содержать заголовок. Изображение должно занимать все окно.

Вариант 5

Написать Windows-приложение — графическую иллюстрацию аппроксимации методом наименьших квадратов зависимости

$$y = a \cdot x + b \cdot x + c \cdot \log_2 x.$$

Создать меню с командами Open, Coefficients, Show, About, Exit.

Команда Exit завершает работу приложения. Команда About открывает окно с информацией о разработчике. Для выбора файла исходных данных (команда Open) использовать объект `OpenFileDialog`. Исходные данные для аппроксимации — массивы экспериментальных значений аргумента x и функции $y(x)$ — сформировать самостоятельно.

При выборе команды Coefficients выводится окно сообщений с вычисленными коэффициентами a, b и c. При выборе команды Show в главном окне приложения отображаются график зависимости и исходные данные в виде точек. Окно должно содержать заголовок. Изображение должно занимать все окно.

Приложение
Спецификаторы формата для строк C#

Спецификаторы используются для форматирования арифметических типов при их преобразовании в строковое представление.

Спецификатор	Описание
C или c	Вывод значений в денежном (currency) формате. По умолчанию перед выводимым значением подставляется символ доллара ($). Этот символ, а также заданное по умолчанию количество позиций можно изменить при помощи объекта `NumberFormatInfo`. Непосредственно после спецификатора можно задать целое число, определяющее длину дробной части
D или d	Вывод целых значений. Непосредственно после спецификатора можно задать целое число, определяющее ширину поля вывода. Недостающие места заполняются нулями, например, вывод числа 12 по формату D3 выглядит как 012
E или e	Вывод значений в экспоненциальном формате, то есть в виде d.ddd...E+ddd или d.ddd...e+ddd. Непосредственно после спецификатора можно задать целое число, определяющее длину дробной части. Минимальная длина порядка — 3 символа
F или f	Вывод значений с фиксированной точностью. Непосредственно после спецификатора можно задать целое число, определяющее длину дробной части, например, число 0,12, выведенное по формату F3, выглядит как 0,120
G или g	Формат общего вида. Применяется для вывода значений с фиксированной точностью или в экспоненциальном формате в зависимости от того, какой формат требует меньшего количества позиций. Для различных типов величин по умолчанию используется разная ширина вывода, например, для `single` — 7 позиций, для `byte` и `sbyte` — 3, для `decimal` — 29

продолжение ⇩

(*продолжение*)

Спецификатор	Описание
N или n	Вывод значений в формате d,ddd,ddd.ddd..., то есть группы разрядов разделяются разделителями, соответствующими региональным настройкам. Непосредственно после спецификации можно задать целое число, определяющее длину дробной части
P или p	Вывод числа в процентном формате (число, умноженное на 100, после которого выводится знак %)
R или r	Отмена округления числа при преобразовании в строку. Гарантирует, что при обратном преобразовании в значение того же типа получится то же самое
X или x	Вывод значений в шестнадцатеричном формате. Если используется прописная буква X, то буквенные символы в шестнадцатеричных символах также будут прописными

Пример применения спецификаторов:

```
using System;
namespace ConsoleApplication1
{    class Class1
    {
        static void Main()
        {
            int i = 1234;
            Console.WriteLine( i.ToString( "C" ) );
            Console.WriteLine( i.ToString( "D5" ) );
            Console.WriteLine( i.ToString( "E" ) );
            Console.WriteLine( i.ToString( "G" ) );
            Console.WriteLine( "{0,9:n2}", i );
            Console.WriteLine( "{0,1:p3}", i );
            Console.WriteLine( "{0,1:x}", i );
        }
    }
}
```

Результат работы программы:

```
1 234,00р.
01234
1,234000E+003
1234
 1 234,00
123 400,000%
4d2
```

Другой пример использования спецификаторов приведен на с. 146.

Список литературы

1. *Биллиг В. А.* Основы программирования на C#. — М.: Изд-во «Интернет-университет информационных технологий — ИНТУИТ.ру», 2006. — 488 с.
2. *Брукс Ф.* Мифический человеко-месяц, или как создаются программные комплексы. — М.: Символ-Плюс, 2000. — 304 с.
3. *Ватсон К.* C#. — М.: Лори, 2004. — 880 с.
4. *Вирт Н.* Алгоритмы и структуры данных. — СПб: Невский диалект, 2001. — 352 с.
5. *Гиббонз П.* Платформа .NET для Java-программистов. — СПб.: Питер, 2003. — 336 с.
6. *Голуб А. И.* C и C++. Правила программирования. — М.: Бином, 1996. — 272 с.
7. *Гуннерсон Э.* Введение в C#. Библиотека программиста. — СПб.: Питер, 2001. — 304 с.
8. *Кораблев В.* Самоучитель Visual C++ .NET. — СПб.: Питер; Киев: Издательская группа BHV, 2004. — 528 с.
9. *Либерти Д.* Программирование на C#. — СПб.: Символ-Плюс, 2003. — 688 с.
10. *Майо Д.* C#. Искусство программирования: Энциклопедия программиста. — Киев: ДиаСофт, 2002. — 656 с.
11. *Майо Дж.* C# Builder. Быстрый старт. — М.: Бином, 2005. — 384 с.
12. *Микелсен К.* Язык программирования C#. Лекции и упражнения: Учебник. — Киев: ДиаСофт, 2002. — 656 с.
13. *Оньон Ф.* Основы ASP.NET с примерами на C#. — М.: Издательский дом «Вильямс», 2003. — 304 с.
14. *Павловская Т. А.* C/C++. Программирование на языке высокого уровня: Учебник для вузов. — СПб.: Питер, 2001. — 464 с.
15. *Павловская Т. А.* Паскаль. Программирование на языке высокого уровня: Учебник для вузов. — СПб.: Питер, 2003. — 393 с.

16. *Паппас К., Мюррей У.* Эффективная работа: Visual C++ .NET. — СПб.: Питер, 2002. — 816 с.
17. *Петцольд Ч.* Программирование для MS Windows на C#. Т. 1. — М.: Издательско-торговый дом «Русская Редакция», 2002. — 576 с.
18. *Петцольд Ч.* Программирование для MS Windows на C#. Т. 2. — М.: Издательско-торговый дом «Русская Редакция», 2002. — 624 с.
19. *Понамарев В. А.* Программирование на C++/C# в Visual Studio .NET 2003. Серия «Мастер программ». — СПб.: БХВ-Петербург, 2004. — 352 с.
20. *Прайс Д., Гандэрлой М.* Visual C#.NET. Полное руководство. — Киев: Век, 2004. — 960 с.
21. *Робинсон С., Корнес О., Глинн Дж.* и др. C# для профессионалов. В 2 т. — М.: Лори, 2003. — 512 с.
22. *Робисон У.* C# без лишних слов. — М.: ДМК Пресс, 2002. — 352 с.
23. *Секунов Н.* Разработка приложений на C++ и C#. Библиотека программиста. — СПб.: Питер, 2003. — 608 с.
24. *Секунов Н.* Самоучитель C#. Серия "Самоучитель". — СПб.: БХВ-Петербург, 2001. — 576 с.
25. *Смайли Д.* Учимся программировать на C# вместе с Джоном Смайли. — Киев: «ДиаСофт», 2003. — 528 с.
26. *Тай Т., Лэм Х. К.* Платформа .NET. Основы. — СПб.: Символ-Плюс, 2003. — 336 с.
27. *Троелсен Э.* C# и платформа .NET. Библиотека программиста. — СПб.: Питер, 2002. — 796 с.
28. *Фролов А. В., Фролов Г. В.* Язык C#: Самоучитель. — М.: Диалог-МИФИ, 2003. — 560 с.
29. *Шилдт Г.* C#: Учебный курс. — СПб.: Питер, 2002. — 512 с.: ил.
30. *Шилдт Г.* Полный справочник по C#. — М.: Издательский дом «Вильямс», 2004. — 752 с.
31. Microsoft Corporation. Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. Учебный курс. Сертификационный экзамен № 70-306, 70-316. — М.: Издательско-торговый дом «Русская Редакция», 2003. — 512 с.
32. Microsoft Corporation. Разработка Web-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. Учебный курс MCAD/MCSD. — М.: Издательско-торговый дом «Русская Редакция», 2003. — 704 с.
33. Microsoft Corporation. Разработка Web-сервисов XML и серверных компонентов на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. Учебный курс MCAD/ MCSD. — М.: Издательско-торговый дом «Русская Редакция», 2004. — 576 с.

Алфавитный указатель

A—C

D—F

G—J

L—N

O—R

S—U

V—Z

А—В

Г—З

И—К

Л—Н

О—П

Р—Т

У—Я

Учебник для вузов

C#. Программирование на языке высокого уровня

Татьяна Александровна Павловская — профессор кафедры информатики и прикладной математики Санкт-Петербургского государственного университета информационных технологий, механики и оптики (СПбГУИТМО), автор нескольких учебников по программированию.

Задача этой книги — кратко, доступно и строго изложить основы C#, одного из самых перспективных современных языков программирования. Книга содержит описание версии C# 2.0 (2005 года) и предназначена для студентов, изучающих язык «с нуля», но будет полезна и опытным программистам, желающим быстро освоить новый язык. Многочисленные достоинства языка C# позволяют расценивать его как перспективную замену языков Паскаль, BASIC и C++ при обучении программированию как студентов, так и школьников старших классов.

В каждой ключевой теме учебника приводится по 20 однотипных вариантов заданий для лабораторных работ. Планируется интернет-поддержка книги по адресу http://ips.ifmo.ru.

Темы, рассмотренные в книге:

- основные понятия платформы .NET;
- типы данных и конструкции языка C#;
- принципы структурного и объектно-ориентированного программирования;
- динамические структуры данных и их реализация в библиотеке .NET;
- классы, интерфейсы, делегаты, события, исключения, сборки, рефлексия типов;
- классы-прототипы, итераторы, указатели, потоки, регулярные выражения;
- основы программирования под Windows.

ПИТЕР

Заказ книг:
Санкт-Петербург
тел.: (812) 703-73-74, postbook@piter.com

www.piter.com —
вся информация о книгах и веб-магазин

ISBN 978-5-496-00861-7